Chapter 11

Compilers and Language Translation

INVITATION TO

Computer Science

6TH EDITION

Objectives

After studying this chapter, students will be able to:

- List the phases of a typical compiler and describe the purpose of each phase
- Demonstrate how to break up a string of text into tokens
- Understand grammar rules written in BNF and use them to parse statements, drawing parse trees for them
- Explain the importance of recursive definitions and avoiding ambiguity in grammar rules
- Explain how semantic analysis uses semantic records to determine meaning

Objectives (continued)

After studying this chapter, students will be able to:

- Show what a code generator would do, given a simple parse tree from one of the book's example grammars
- Explain the historical importance of code optimization, and why it seems less central today
- List and explain the local optimizations discussed here
- Describe the approach of global optimization, illustrating it with an example

Introduction

- High-level languages must be translated into machine instructions
- Compilers do the translation
- Compiler-writing is difficult and complex
- Compilers must be:
  - Correct: machine instructions must do exactly what the high-level instructions mean
  - Efficient and concise: code produced should be optimized and run fast

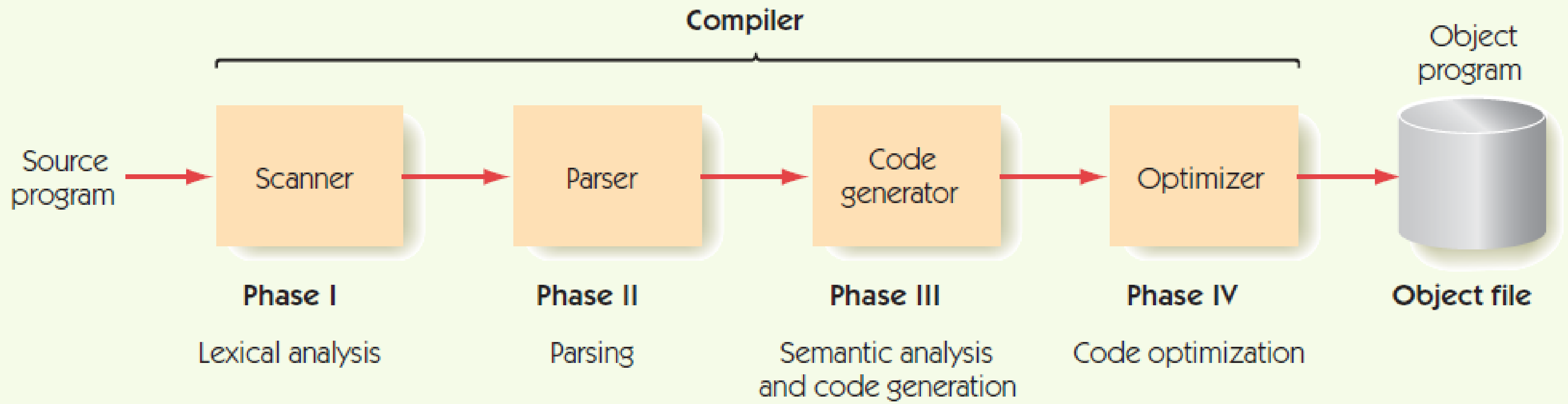
The Compilation Process

Four phases of compilation:

- Phase I: Lexical analysis
  - Groups characters into tokens (equivalent to words)
- Phase II: Parsing
  - Checks grammatical structure and builds internal representation of program
- Phase III: Semantic Analysis and Code Generation
  - Analyze meaning and generate machine instructions
- Phase IV: Code Optimization
  - Improve efficiency of code in time or space required

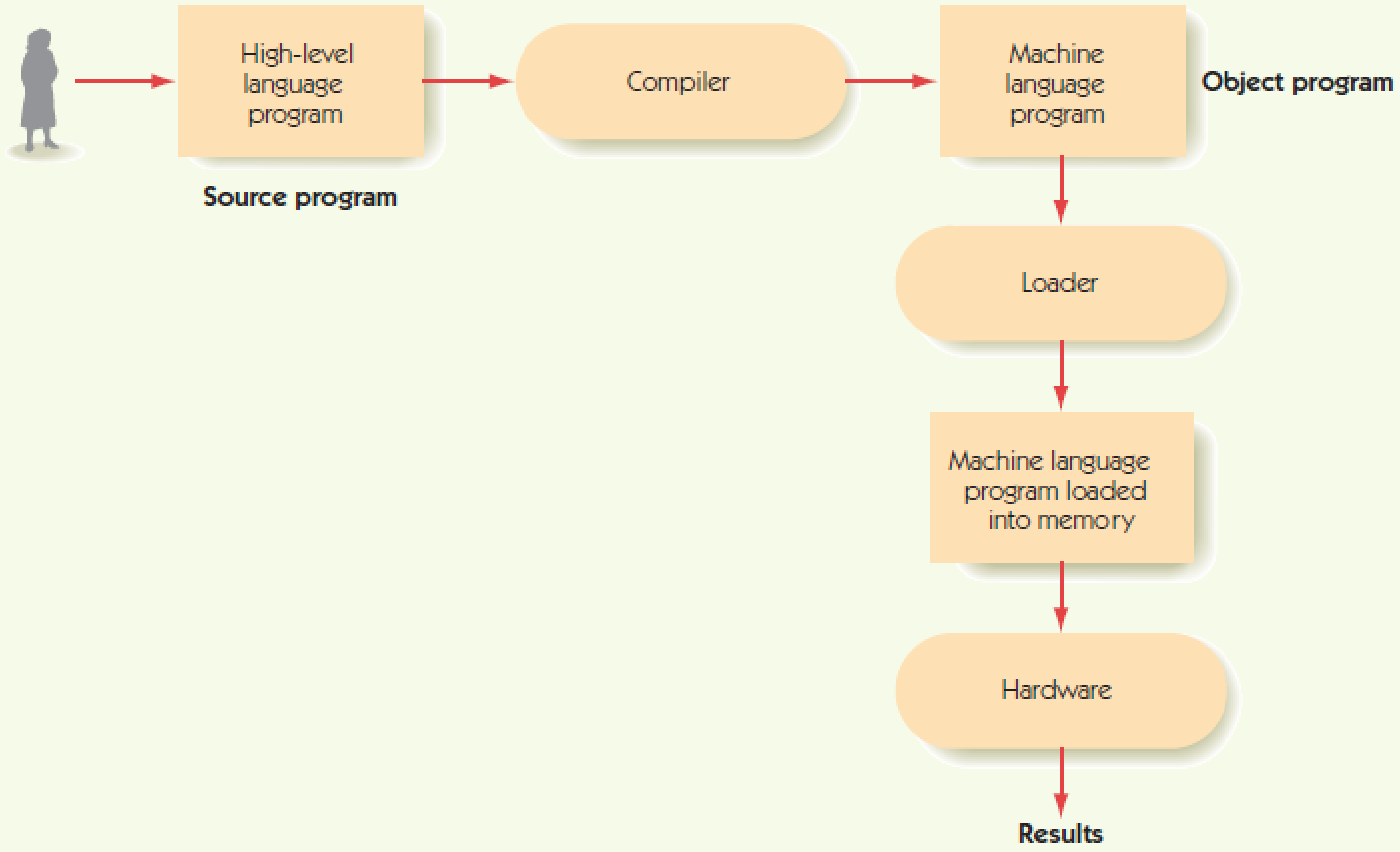
The Compilation Process (continued)

FIGURE 11.1



General structure of a compiler

FIGURE 11.2



Overall execution sequence of a high-level language program

The Compilation Process

Phase I: Lexical Analysis (continued)

- A **Lexical analyzer** or **scanner**
  - Groups input characters into **tokens**
  - Discards unneeded characters
    - E.g., blanks, tabs, comment text
  - Determines the type of each token
    - E.g., symbol, number, left parenthesis
- Tokens are words and punctuation that are meaningful to the language

The Compilation Process
Phase I: Lexical Analysis (continued)

FIGURE 11.3

Token Type	Classification Number
symbol	1
number	2
=	3
+	4
−	5
;	6
==	7
if	8
else	9
(	10
)	11

Typical token classifications

The Compilation Process
Phase II: Parsing (continued)

- A **parser** takes a list of tokens and
  - Determines grammatical structure
  - Diagrams the program, building a **parse tree**
- **Syntax** = grammatical structure
- Syntax is defined by **rules** (**productions**)
  - **BNF (Backus-Naur Form)** is notation for describing rules
- A **grammar** is the set of rules that define a language

The Compilation Process
Phase II: Parsing (continued)

BNF

- Rules:

  left-hand side ::= right-hand side

  <sentence> ::= <subject> <verb> <object>

- Left-hand side: grammatical category
- Right-hand side:
  - pattern that captures the structure of category

The Compilation Process
Phase II: Parsing (continued)

BNF

- Patterns made from **terminals** and **nonterminals**
- Terminals = tokens from lexical analyzer
- Nonterminals = grammatical categories
  – Written in <angle brackets>
- **Goal symbol**
  – final nonterminal
  – means complete grammatical program is found
- **Metasymbols**: <, >, ::=, |, Λ
  – Λ (lambda) is **null string** = emptiness

The Compilation Process
Phase II: Parsing (continued)

"If, by repeated applications of the rules of the grammar, 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 it cannot convert the input tokens into the goal symbol, then this is not a syntactically valid statement of the language."

The Compilation Process
Phase II: Parsing (continued)

Parsing example:

x = y + z

Apply <variable> ::= … rules

<variable> = <variable> + <variable>

Apply <expression> ::= <variable> + <variable> rule

<variable> = <expression>

Apply <assignment statement> rule

<assignment statement>

The Compilation Process

Phase II: Parsing (continued)

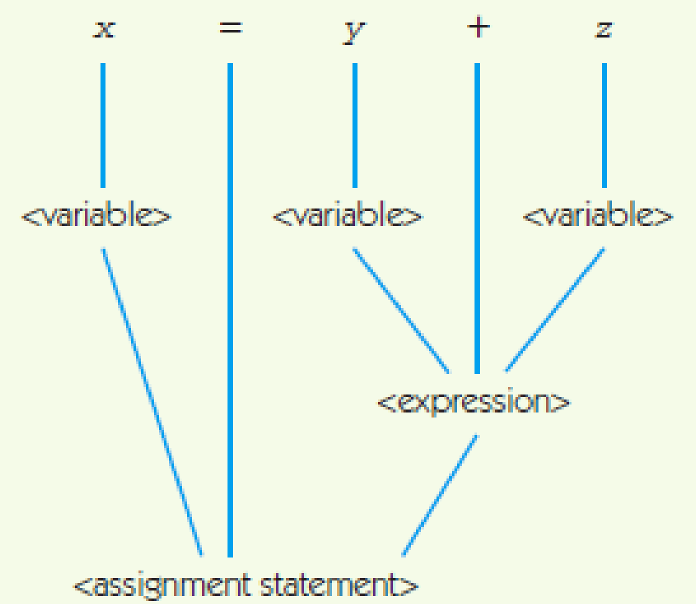
FIGURE 11.4

Number	**Rule**
1	<assignment statement> ::= <variable> = <expression>
2	<expression> ::= <variable> | <variable> + <variable>
3	<variable> ::= *x* | *y* | *z*

First attempt at a grammar for a simplified assignment statement

The Compilation Process
Phase II: Parsing (continued)

FIGURE 11.5



Parse tree produced by the parser

The Compilation Process
Phase II: Parsing (continued)

- **Look-ahead parsing algorithms**
  - Look at future tokens to choose the right rule to apply
- Allowing for arbitrary-length patterns
  - x = x + y + z + q + p
- **Recursive definition**: definition includes left-hand term on the right-hand side:
  - <expression> ::= <expression> + <expression>

The Compilation Process

Phase II: Parsing (continued)

FIGURE 11.6

Number	**Rule**
1	<assignment statement> ::= <variable> = <expression>
2	<expression> ::= <variable> | <expression> + <expression>
3	<variable> ::= *x* | *y* | *z*

Second attempt at a grammar for assignment statements

The Compilation Process
Phase II: Parsing (continued)

- **Ambiguous**: a grammar where the same string can be parsed multiple ways

x = x – y – z means x = (x – y) – z

OR

x = x – y – z means x = x – (y – z)

- Grammars must be unambiguous!

The Compilation Process
Phase II: Parsing (continued)

FIGURE 11.8

Number	Rule
1	<assignment statement> ::= <variable> = <expression>
2	<expression> ::= <variable> | <expression> + <variable>
3	<variable> ::= *x* | *y* | *z*

Third attempt at a grammar for assignment statements

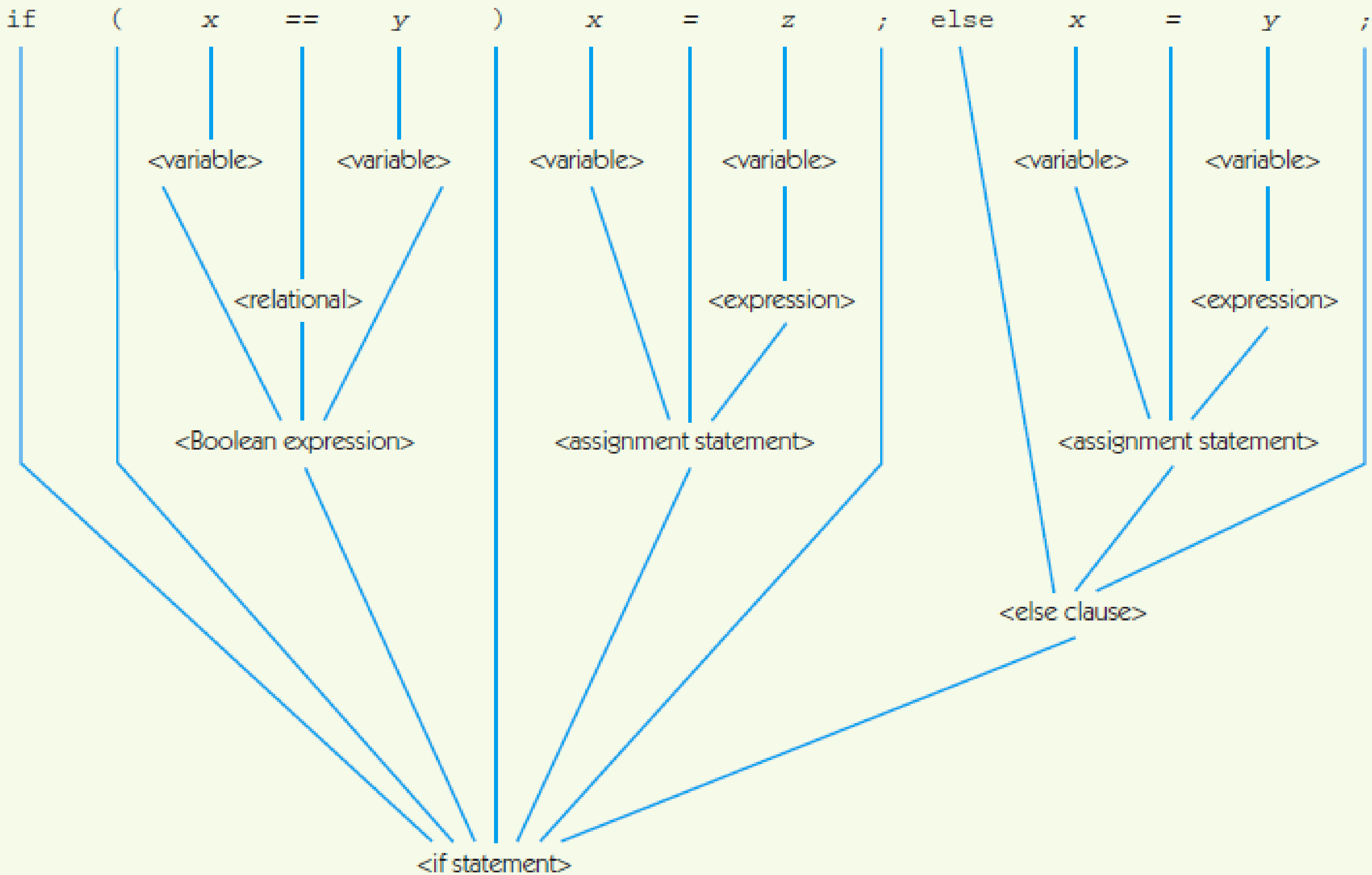
The Compilation Process
Phase II: Parsing (continued)

FIGURE 11.9

<table>
<tr><th>Number</th><th>Rule</th></tr>
<tr><td>1</td><td><if statement> ::= if (<Boolean expression>) <assignment statement> ;
<else clause></td></tr>
<tr><td>2</td><td><Boolean expression> ::= <variable> | <variable> <relational> <variable></td></tr>
<tr><td>3</td><td><relational> ::= == | < | ></td></tr>
<tr><td>4</td><td><variable> ::= x | y | z</td></tr>
<tr><td>5</td><td><else clause> ::= else <assignment statement> ; | Λ</td></tr>
<tr><td>6</td><td><assignment statement> ::= <variable> = <expression></td></tr>
<tr><td>7</td><td><expression> ::= <variable> | <expression> + <variable></td></tr>
</table>

Grammar for a simplified version of an *if-else* statement

FIGURE 11.10



Parse tree for the statement if (x == y) x = z; else x = y;

The Compilation Process
Phase III: Semantics and Code Generation

Semantic analysis

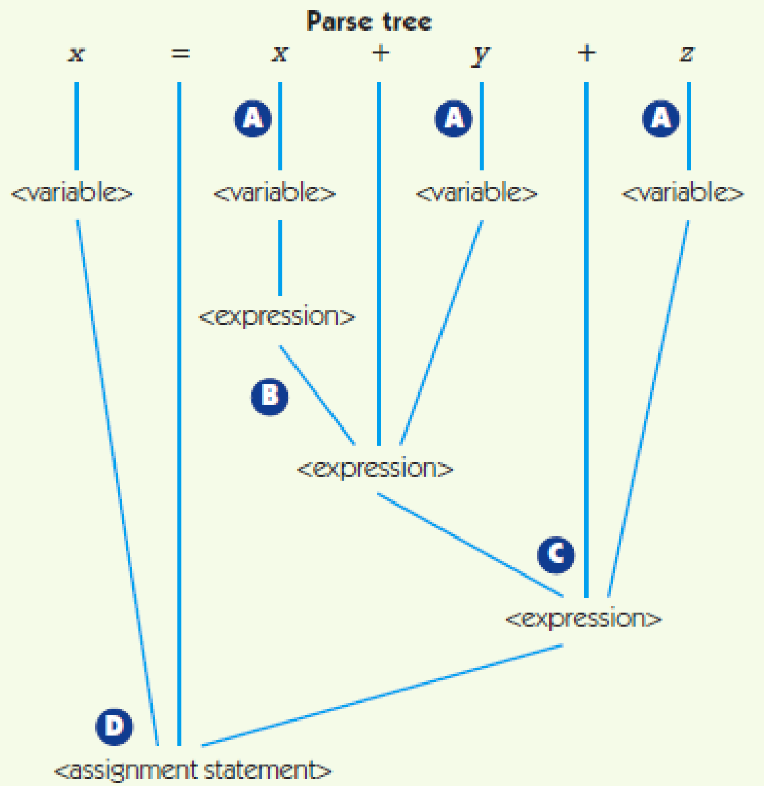
- Check that parse tree all makes sense
- Grammatical statements can be meaningless
  - E.g., Bees bark
- **Semantic records**
  - Store information about actual values associated with nonterminals
  - <variable> came from a token “x” and its type was integer

The Compilation Process
Phase III: Semantics and Code Generation (continued)

Code generation

- Translate parse tree pieces to assembly code
- Can build semantic records as it goes
- Semantic analysis simultaneous with generation
- Not all parts of parse tree produce code

FIGURE 11.11



Generated code

```
--Here is the code for the production labeled B
        LOAD    X
        ADD     Y
        STORE   TEMP     --Temp holds the expression (x + y)
--Here is the code for the production labeled C
        LOAD    TEMP
        ADD     Z
        STORE   TEMP2    --Temp2 holds (x + y + z)
--Here is the code for the production labeled D
        LOAD    TEMP2
        STORE   X        --X now holds the correct result
                         --The remainder of the program goes here
--These next three pseudo-ops are generated by the productions labeled A
X:      .DATA   0
Y:      .DATA   0
Z:      .DATA   0
--The pseudo-ops for these temporary variables are generated
by productions B and C
TEMP:   .DATA   0
TEMP2:  .DATA   0
```

Code generation for the assignment statement $x = x + y + z$

The Compilation Process
Phase IV: Code Optimization

- **Code optimization**
  - Improving the time or space efficiency of code produced by a compiler
- Early days: “Hardware is expensive, programmers are cheap”
  - Humans could write more optimal code than a compiler
- These days: “Hardware is cheap, programmers are expensive”

The Compilation Process
Phase IV: Code Optimization (continued)

- Modern compilers optimize, but focus on other issues:
  - **Visual development environments** help programmers see what is happening
  - **Online debuggers** help find and correct bugs
  - Reusable code libraries and toolkits

The Compilation Process

Phase IV: Code Optimization (continued)

Local optimization

- Examine small chunks of assembly code (< 5 instructions)
  - **Constant evaluation**, compute arithmetic expressions at compile-time if possible
  - **Strength reduction**, use faster arithmetic alternatives (e.g., 2 * x is equivalent to x + x)
  - **Eliminating unnecessary operations**, remove operations that are unneeded, like a LOAD of a value already in memory

The Compilation Process

Phase IV: Code Optimization (continued)

Global optimization

- Looks at large segments of the program
- Blocks like while loops, if statements, and procedures
- Much more difficult, much bigger effect!

Optimization cannot overcome an inefficient algorithm

The Compilation Process

Phase IV: Code Optimization (continued)

FIGURE 11.12

```
        LOAD    X
        ADD     Y
        ADD     Z
        STORE   X       --X now holds the correct result
        .
        .               --The remainder of the program goes here
        .
X:      .DATA   0
Y:      .DATA   0
Z:      .DATA   0
```

Optimized code for the assignment statement $x = x + y + z$

Summary

- High-level languages require compilers to translate programs into assembly language
- Compilation is much more difficult and complex than assemblers translating assembly to machine instructions
- Compiler phases include: lexical analysis, parsing, semantic analysis, code generation, and code optimization
- Lexical analysis converts a text of characters into a list of tokens

Summary (continued)

- Parsing is based on a formal grammar specifying the rules for a language
- Parsers check grammaticality and build parse trees
- Semantic analysis checks parse trees for meaning: can code meaningfully be generated
- Code generation produces assembly instructions from the parse tree
- Code optimization looks for opportunities to make generated code better