Chapter 6:
Programming Languages

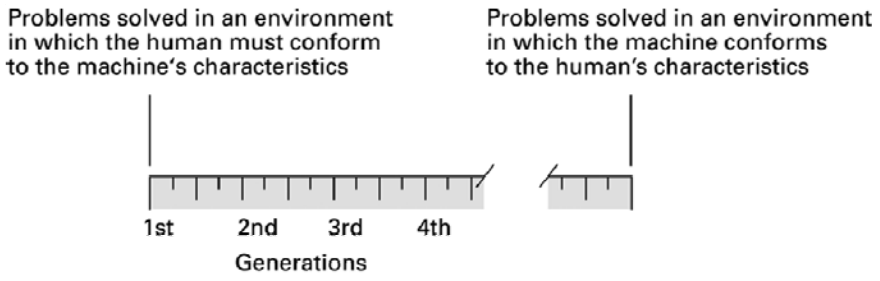
Computer Science: An Overview
Eleventh Edition

by
J. Glenn Brookshear

Chapter 6: Programming Languages

- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

Figure 6.1 Generations of programming languages



Second-generation: Assembly language

- A mnemonic system for representing machine instructions
  - Mnemonic names for op-codes
  - Identifiers: Descriptive names for memory locations, chosen by the programmer

Assembly Language Characteristics

- One-to-one correspondence between machine instructions and assembly instructions
  - Programmer must think like the machine
- Inherently machine-dependent
- Converted to machine language by a program called an **assembler**

Program Example

Machine language	Assembly language
156C	LD R5, Price
166D	LD R6, ShipCharge
5056	ADDI R0, R5 R6
30CE	ST R0, TotalCost
C000	HLT

Third Generation Language

- Uses high-level primitives
  - Similar to our pseudocode in Chapter 5
- Machine independent (mostly)
- Examples: FORTRAN, COBOL
- Each primitive corresponds to a sequence of machine language instructions
- Converted to machine language by a program called a **compiler**

Figure 6.2 The evolution of programming paradigms

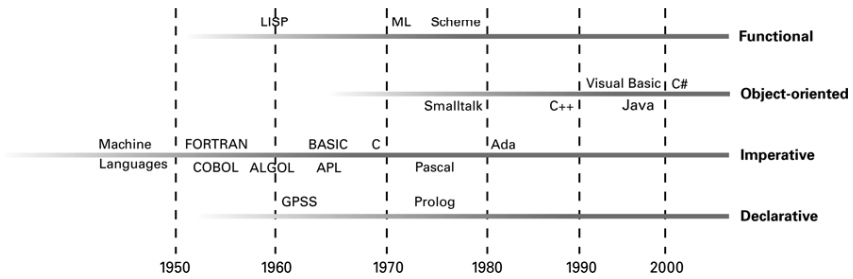


Figure 6.3 A function for checkbook balancing constructed from simpler functions

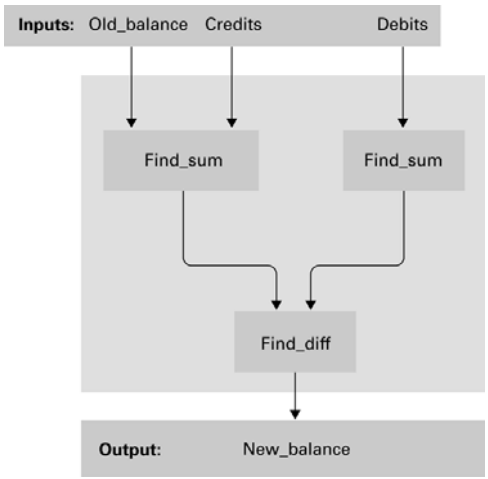
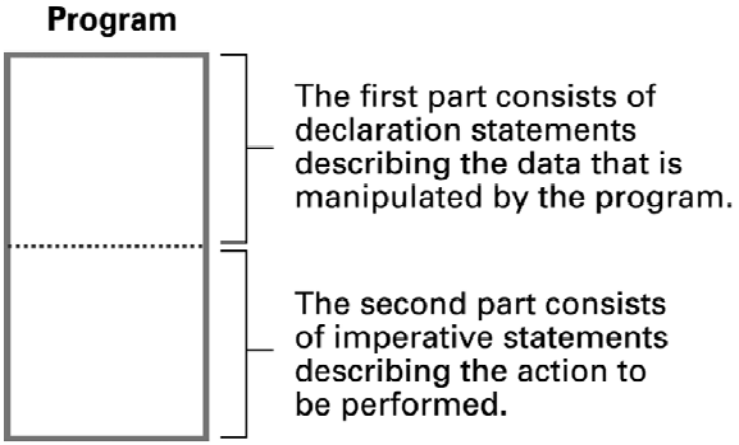


Figure 6.4 The composition of a typical imperative program or program unit



Data Types

- Integer: Whole numbers
- Real (float): Numbers with fractions
- Character: Symbols
- Boolean: True/false

Variable Declarations

```
float    Length, Width;
int      Price, Total, Tax;
char     Symbol;
```

Figure 6.5 A two-dimensional array with two rows and nine columns

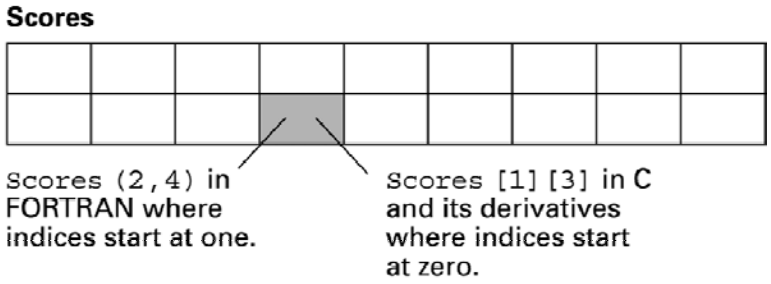


Figure 6.6 The conceptual structure of the aggregate type Employee

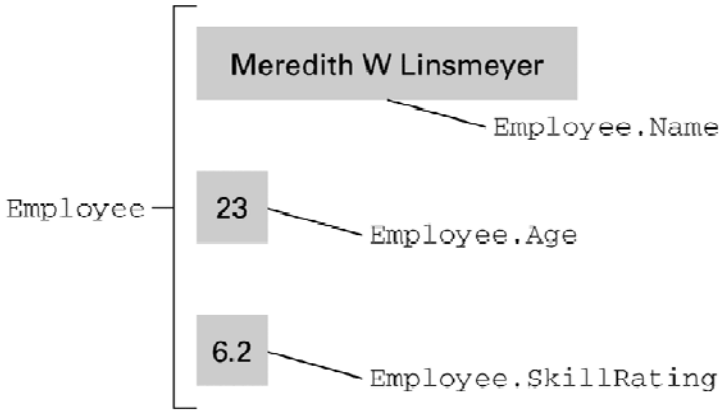
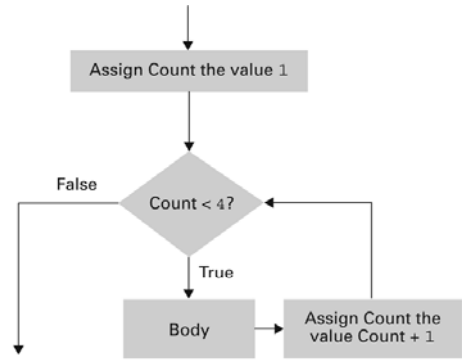


Figure 6.7 The for loop structure and its representation in C++, C#, and Java




```
for (int Count = 1; Count < 4; Count++)
    body ;
```


Procedural Units

- Local versus Global Variables
- Formal versus Actual Parameters
- Passing parameters by value versus reference
- Procedures versus Functions

Figure 6.8 The flow of control involving a procedure

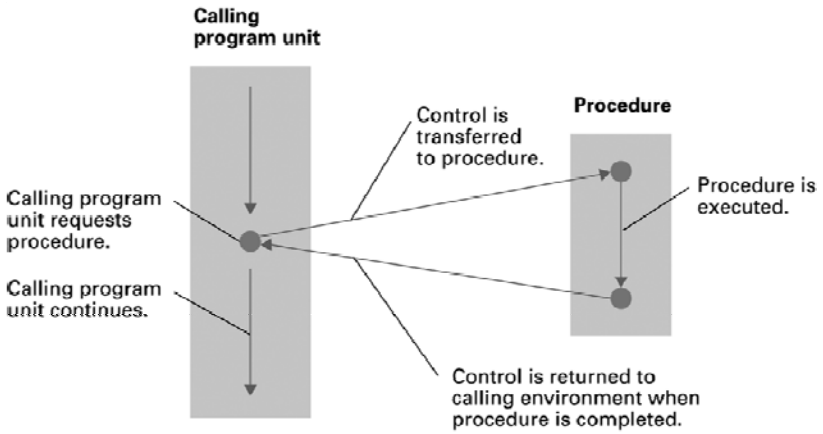


Figure 6.9 The procedure ProjectPopulation written in the programming language C

Starting the head with the term "void" is the way that a C programmer specifies that the program unit is a procedure rather than a function. We will learn about functions shortly.

The formal parameter list. Note that C, as with many programming languages, requires that the data type of each parameter be specified.

```
void ProjectPopulation (float GrowthRate)
{ int Year;
  Population[0] = 100.0;
  for (Year = 0; Year =< 10; Year++)
  Population[Year+1] = Population[Year] + (Population[Year] * GrowthRate);
}
```

This declares a local variable named Year.

These statements describe how the populations are to be computed and stored in the global array named Population.

Figure 6.10
Executing the procedure Demo and passing parameters by value

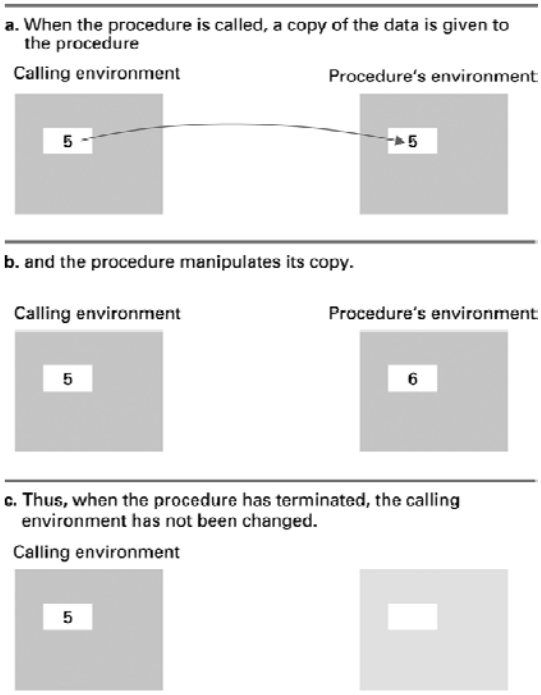


Figure 6.11
Executing the procedure Demo and passing parameters by reference

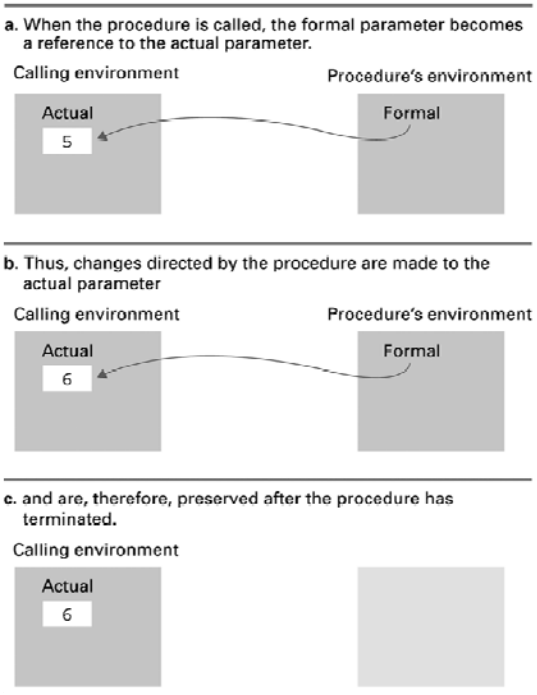


Figure 6.12 The function CylinderVolume written in the programming language C

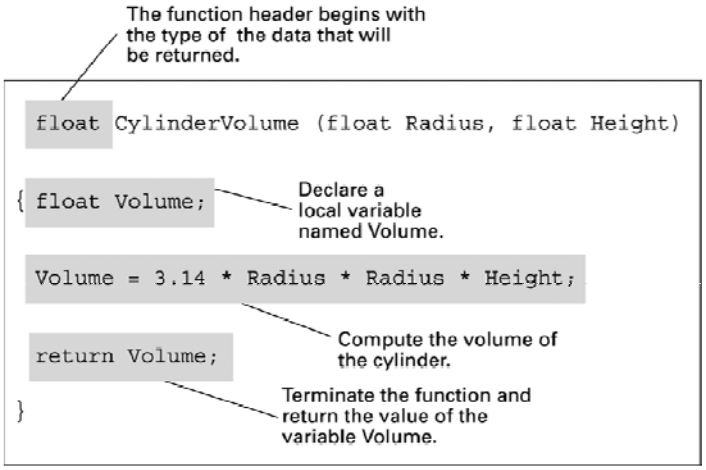


Figure 6.13 The translation process

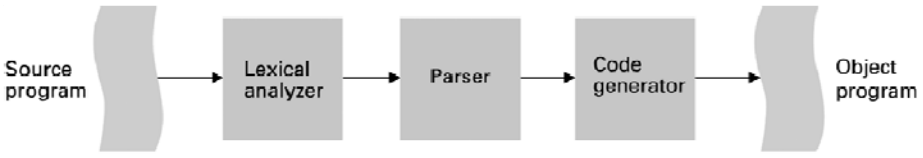


Figure 6.14 A syntax diagram of our if-then-else pseudocode statement

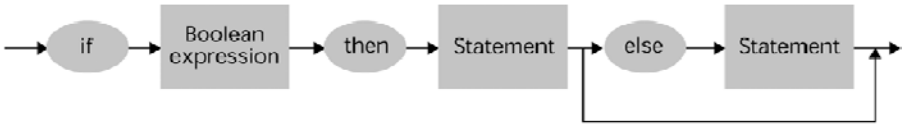


Figure 6.15 Syntax diagrams describing the structure of a simple algebraic expression

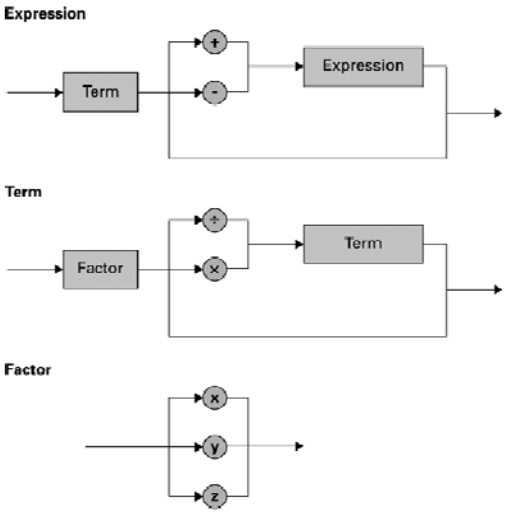


Figure 6.16 The parse tree for the string x + y x z based on the syntax diagrams in Figure 6.17

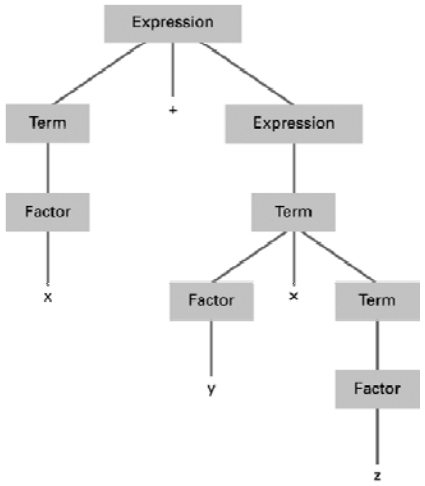


Figure 6.17 Two distinct parse trees for the statement if B1 then if B2 then S1 else S2

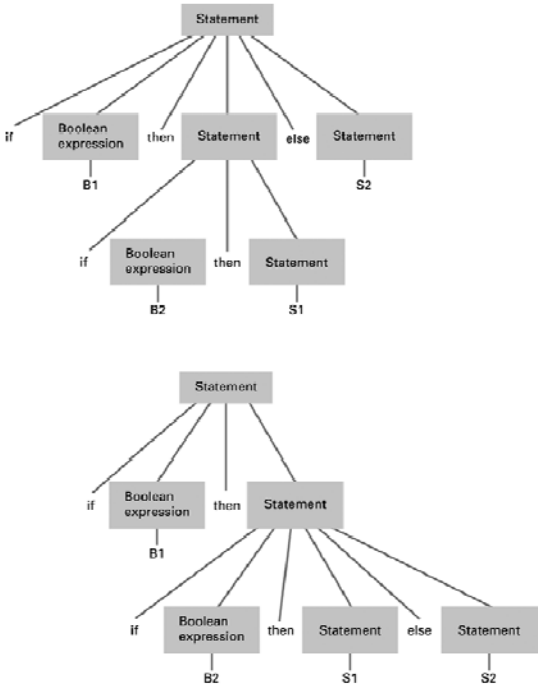
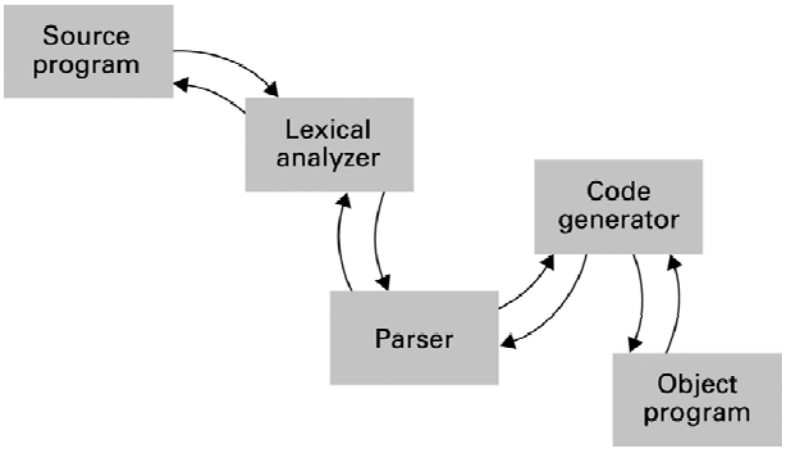


Figure 6.18 An object-oriented approach to the translation process

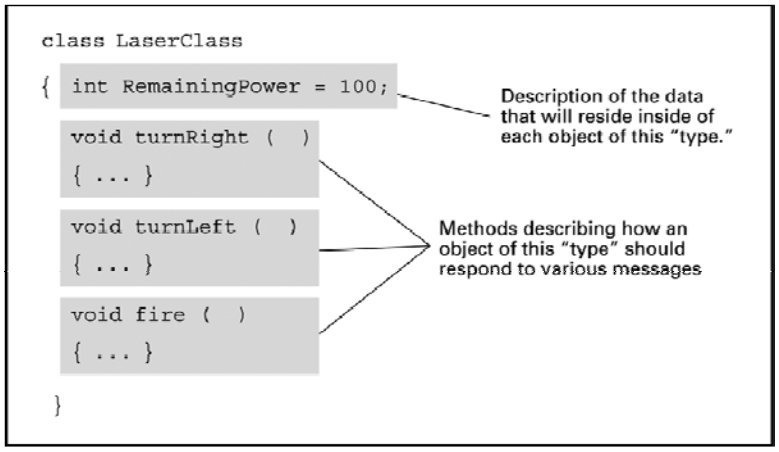


Objects and Classes

- **Object:** Active program unit containing both data and procedures
- **Class:** A template from which objects are constructed

An object is called an **instance** of the class.

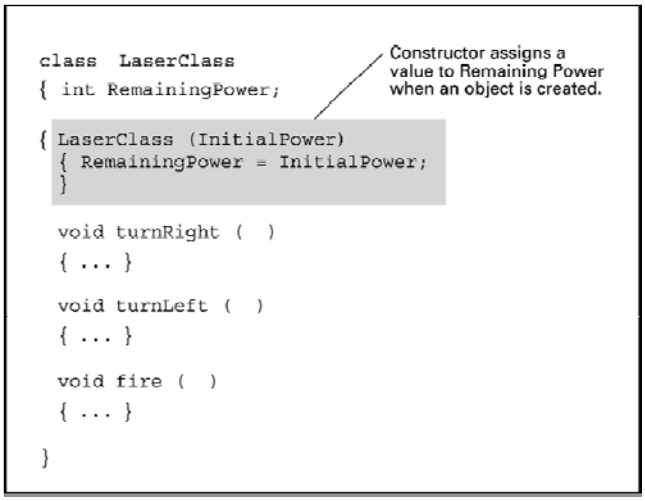
Figure 6.19 The structure of a class describing a laser weapon in a computer game



Components of an Object

- **Instance Variable:** Variable within an object
  - Holds information within the object
- **Method:** Procedure within an object
  - Describes the actions that the object can perform
- **Constructor:** Special method used to initialize a new object when it is first constructed

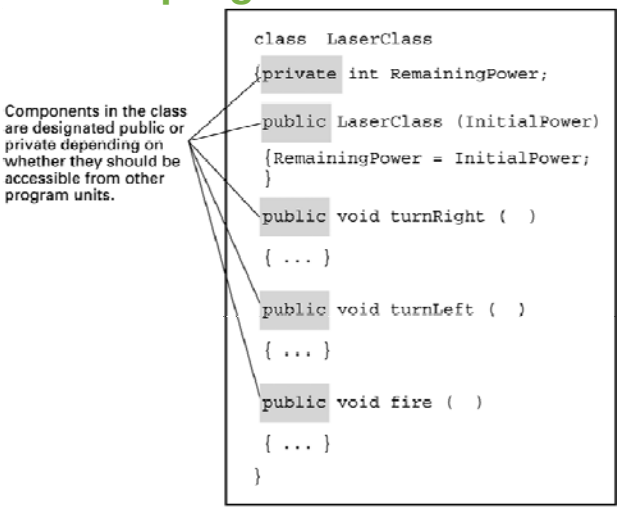
Figure 6.21 A class with a constructor



Object Integrity

- **Encapsulation:** A way of restricting access to the internal components of an object
  - Private
  - Public

Figure 6.22 Our LaserClass definition using encapsulation as it would appear in a Java or C# program



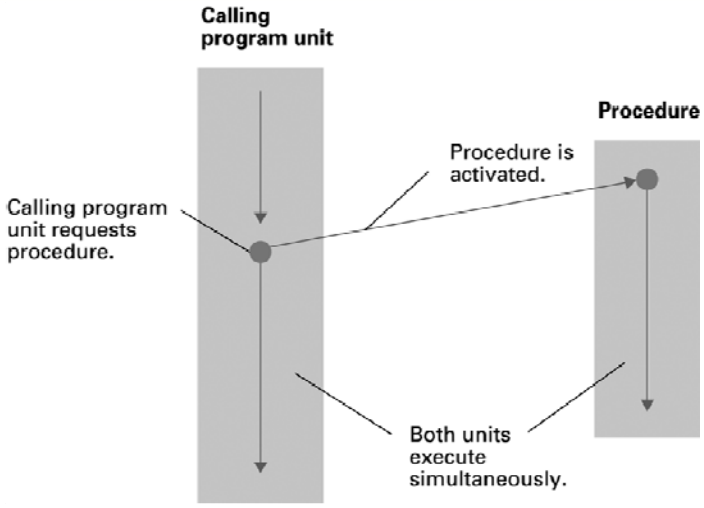
Additional Object-oriented Concepts

- **Inheritance:** Allows new classes to be defined in terms of previously defined classes
- **Polymorphism:** Allows method calls to be interpreted by the object that receives the call

Programming Concurrent Activities

- **Parallel** (or **concurrent**) **processing:** simultaneous execution of multiple processes
  - True concurrent processing requires multiple CPUs
  - Can be simulated using time-sharing with a single CPU

Figure 6.23 Spawning threads



Controlling Access to Data

- **Mutual Exclusion:** A method for ensuring that data can be accessed by only one process at a time
- **Monitor:** A data item augmented with the ability to control access to itself

Declarative Programming

- **Resolution:** Combining two or more statements to produce a new statement (that is a logical consequence of the originals).
  - Example: (P OR Q) AND (R OR ¬Q) resolves to (P OR R)
  - **Resolvent:** A new statement deduced by resolution
  - **Clause form:** A statement whose elementary components are connected by the Boolean operation OR
- **Unification:** Assigning a value to a variable so that two statements become "compatible."

Figure 6.24 Resolving the statements (*P* OR *Q*) and (*R* OR ¬*Q*) to produce (*P* OR *R*)

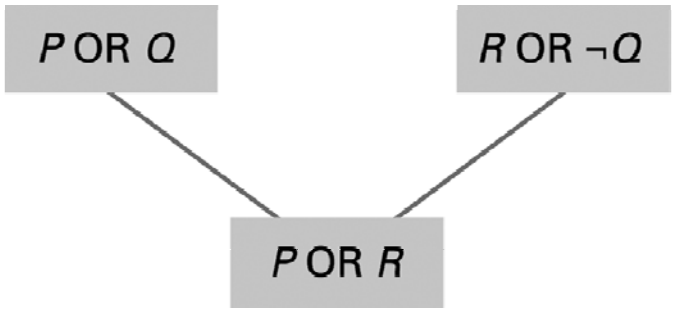
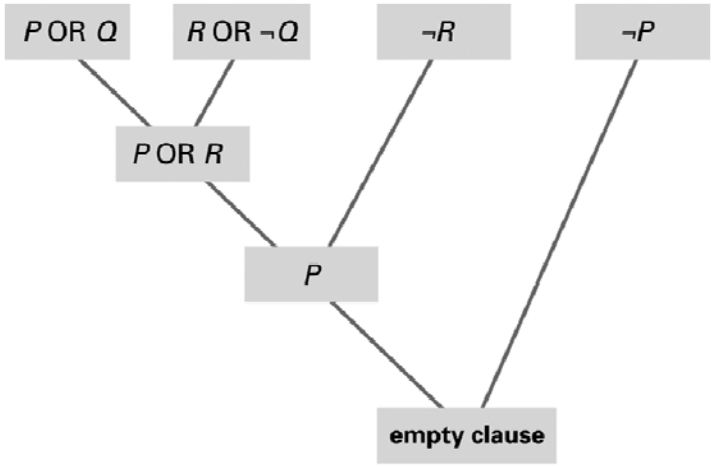


Figure 6.25 Resolving the statements (*P* OR *Q*), (*R* OR ¬*Q*), ¬R, and ¬*P*



Prolog

- **Fact:** A Prolog statement establishing a fact
  - Consists of a single predicate
  - Form: *predicateName*(*arguments*).
    - Example: `parent(bill, mary).`
- **Rule:** A Prolog statement establishing a general rule
  - Form: *conclusion* :- *premise.*
    - :- means “if”
  - Example: `wise(X) :- old(X).`
  - Example: `faster(X,Z) :- faster(X,Y), faster(Y,Z).`