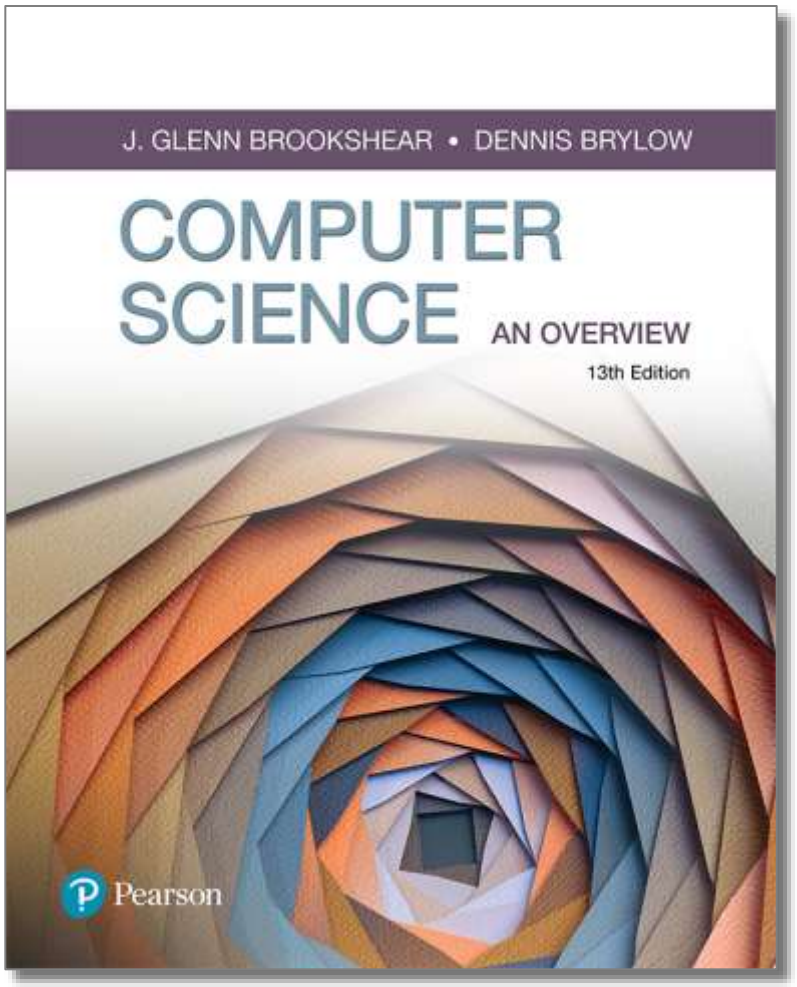
Computer Science An Overview

13th Edition



Chapter 6

Programming Languages

Chapter 6: Programming Languages

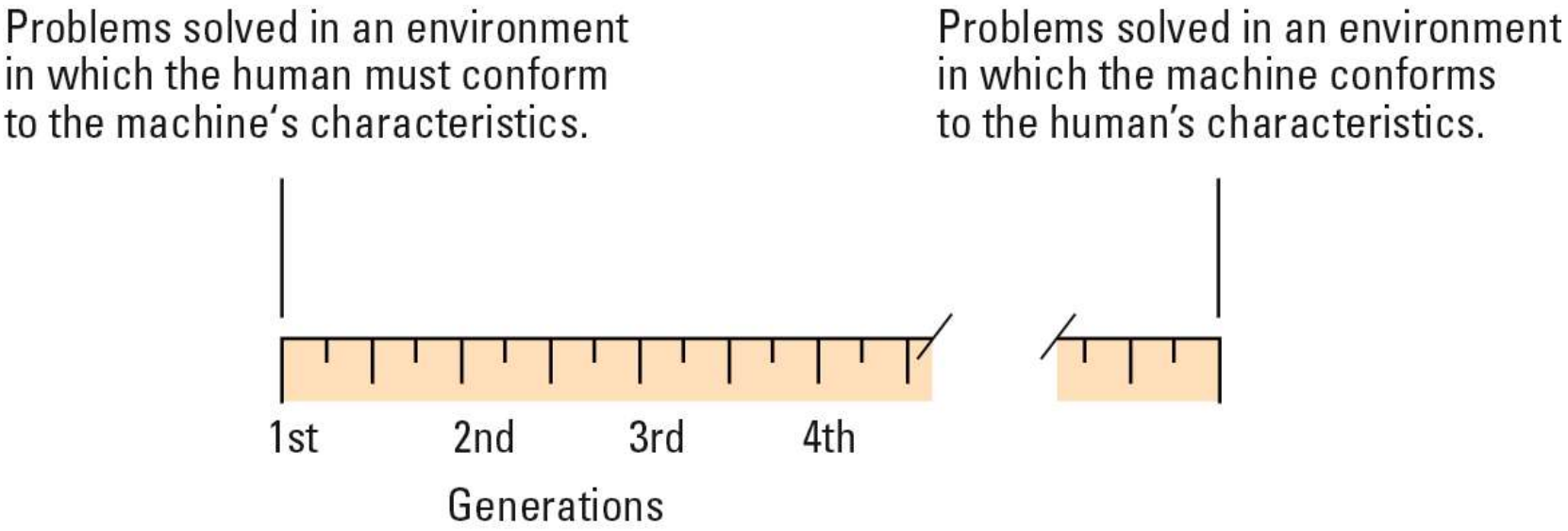
- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

What is a Programming Language?

A programming language is a notational system for describing computation in a machine-readable and human-readable form.

— Louden

Figure 6.1 Generations of programming languages



Generations of Programming Languages

- **Occurring in "generations" or "levels"**
  - **Levels-Machine languages to natural languages**
- **There are five generations :**
  - **Lower level closer to machine language**
  - **Higher level closer to human-like language**

Generation	Sample Statement
First: Machine	1111001001110011110100100001000001110000000101011
Second: Assembly	ADD 210(8, 13),02B(4, 7)
Third: Procedural	if (score > = 90) grade = 'A';
Fourth: Problem	SELECT client FROM dailyLog WHERE serviceEnd > 17
Fifth: Natural and Visual	If patient is dizzy, then check temperature and blood pressure.

6.1 Historical Perspective

- Early Generations
  - Machine Language (e.g. Vole)
  - Assembly Language
- Machine Independent Language
- Beyond – more powerful abstractions

Second-generation: Assembly language

- A mnemonic system for representing machine instructions
  - Mnemonic names for op-codes
  - Program **variables** or **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

```
156C
166D
5056
30CE
C000
```

Assembly language

```
LD    R5, Price
LD    R6, ShipCharge
ADDI  R0, R5 R6
ST    R0, TotalCost
HLT
```

```
;CLEAR SCREEN USING BIOS
CLR: MOV AX,0600H        ;SCROLL SCREEN
     MOV BH,30           ;COLOUR
     MOV CX,0000         ;FROM
     MOV DX,184FH        ;TO 24,79
     INT 10H             ;CALL BIOS;
;INPUTTING OF A STRING
KEY: MOV AH,0AH          ;INPUT REQUEST
     LEA DX,BUFFER       ;POINT TO BUFFER WHERE STRING STORED
     INT 21H             ;CALL DOS
     RET                 ;RETURN FROM SUBROUTINE TO MAIN PROGRAM;
; DISPLAY STRING TO SCREEN
SCR: MOV AH,09           ;DISPLAY REQUEST
     LEA DX,STRING       ;POINT TO STRING
     INT 21H             ;CALL DOS
     RET                 ;RETURN FROM THIS SUBROUTINE;
```

Assembly code

Assembler

```
0001010010110101010101010101010100010
1110110101010101010101110010100010110
0010100101010010111101011101011101010
1001010010110101010101010101010110110
0110100100110010111101011101010100010
0001000101011101010101000101010111010
1010100101010010101101011101011101011
0001010010110101010101010101010100010
```

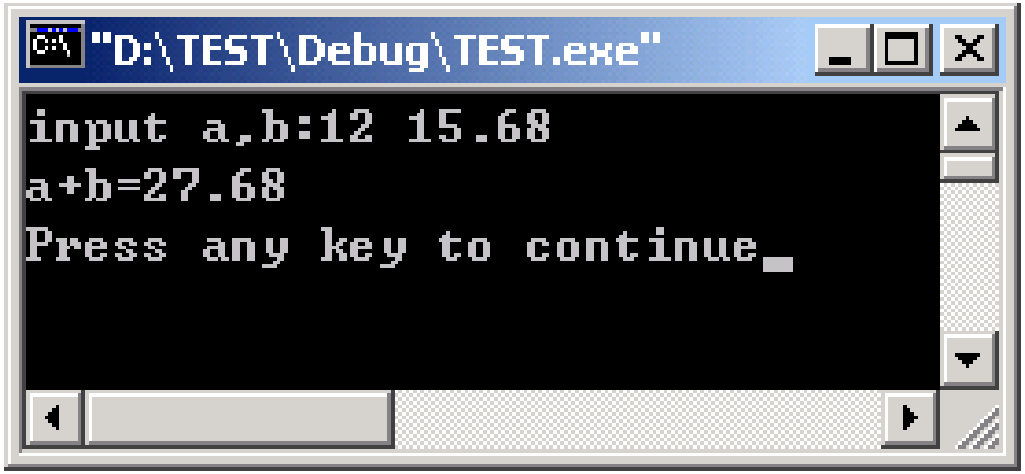
Object code

Third Generation Language

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

C program example

```
#include  "iostream.h"
void main()
{
    int a;
    float b;
    cout<<"input a,b:";
    cin>>a>>b;
    cout<<"a+b="<<a+b<<endl;
}
```



Fourth Generation Language

- SQL

Insert into students (student_ID, student_name, Gender, ACM_member, Major, Date_of_birth, scholarship, score) values ("10010","Maggie","F",No,"Pharmacy",#10/20/1989#,1000,88)

update students set score=55

select student_name, major, scholarship from students

Figure 6.2 The evolution of programming paradigms

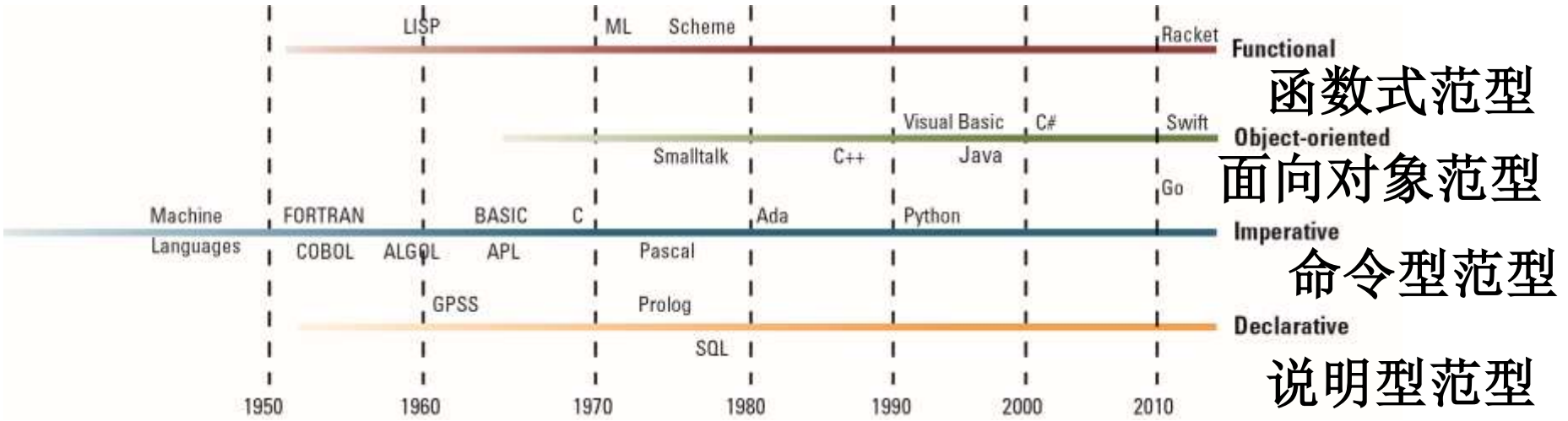
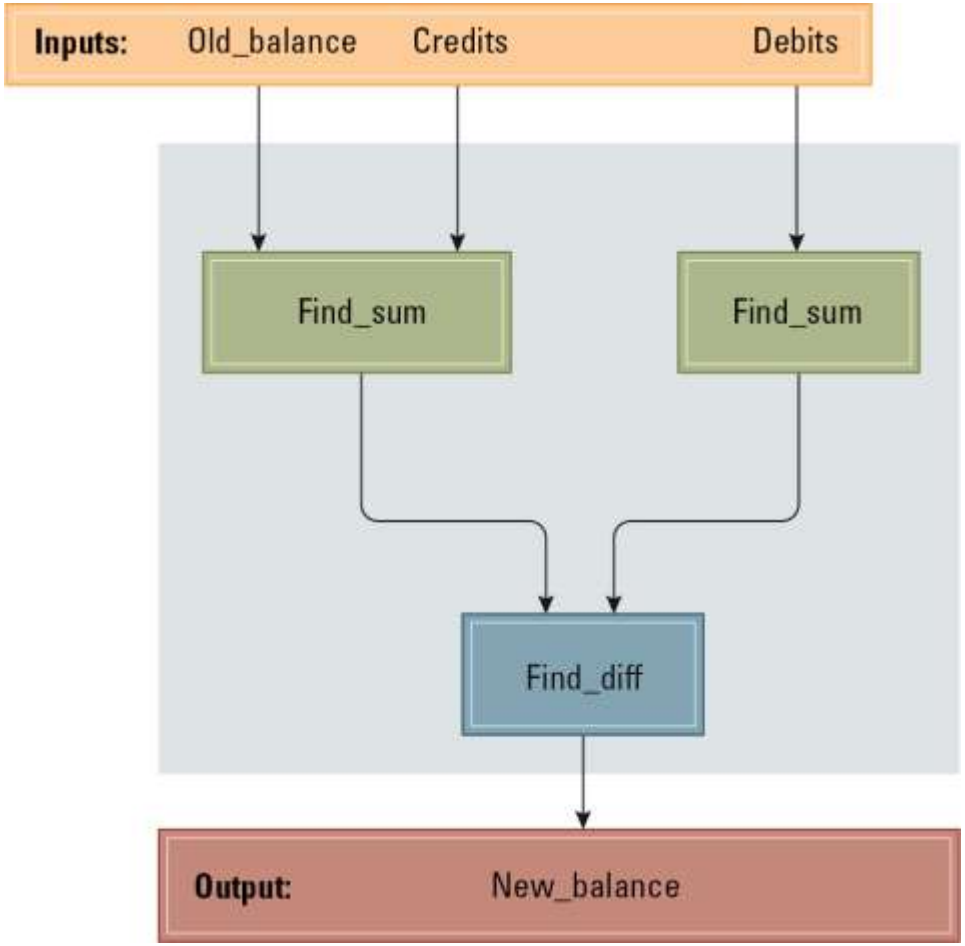


Figure 6.3 A function for checkbook balancing constructed from simpler functions



函数式范型将计算过程表达为一系列函数调用，通过函数来处理数据。程序可以看作是接受输入和产生输出的实体。

面向对象范型

- 面向对象编程（Object-Oriented Programming，简称OOP）是一种编程范型，它将现实世界中的实体抽象为对象，并通过对象之间的交互来设计和构建软件系统。

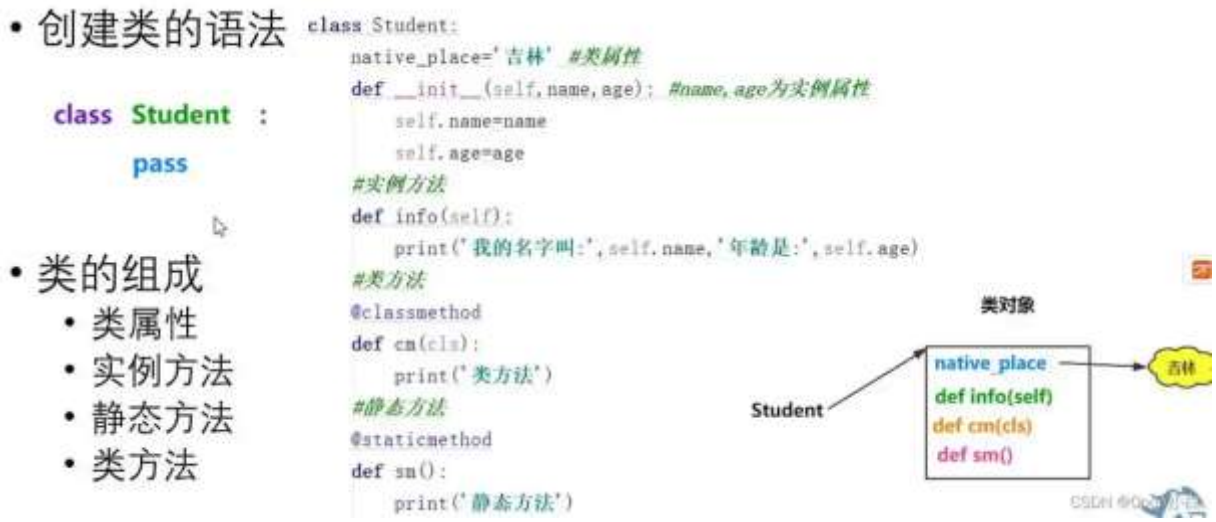
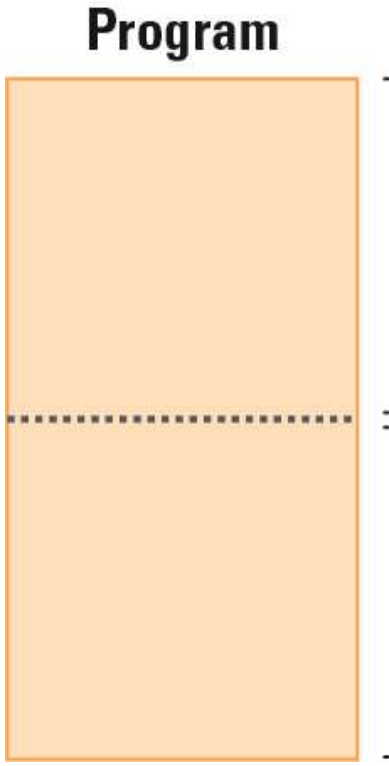


Figure 6.4 The composition of a typical imperative program or program unit




```
#include  "iostream.h"

void main()

{

    int a;

    float b;

    cout<<"input a,b:";

    cin>>a>>b;

    cout<<"a+b="<<a+b<<endl;

}
```

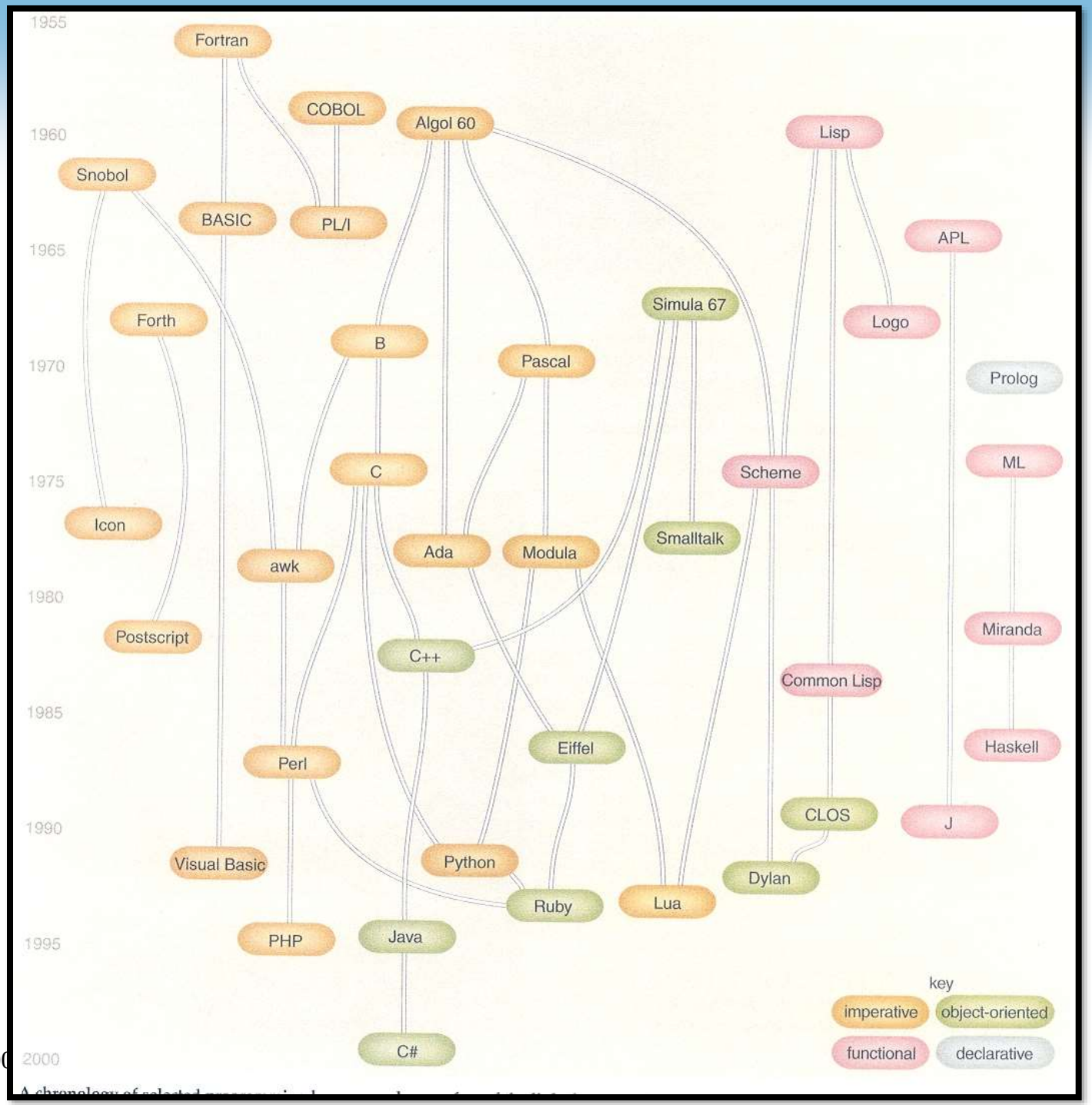
命令型范型

(面向过程)开发一个命令序列,遵照这个序列,对数据进行操作以产生所期望的结果

说明型范型

- 描述要解决的问题，而不是解决该问题的算法，如：SQL
- 它关注“做什么”而不是“怎么做”。在说明型编程中，开发者表达逻辑和计算的结果，而具体的执行细节由编译器或解释器来处理。
  - 查询所有订单，并按订单日期降序排序
  - SELECT * FROM orders ORDER BY order_date DESC;

SQL允许开发者以声明式的方式表达他们的需求，而不需要关心底层的实现细节



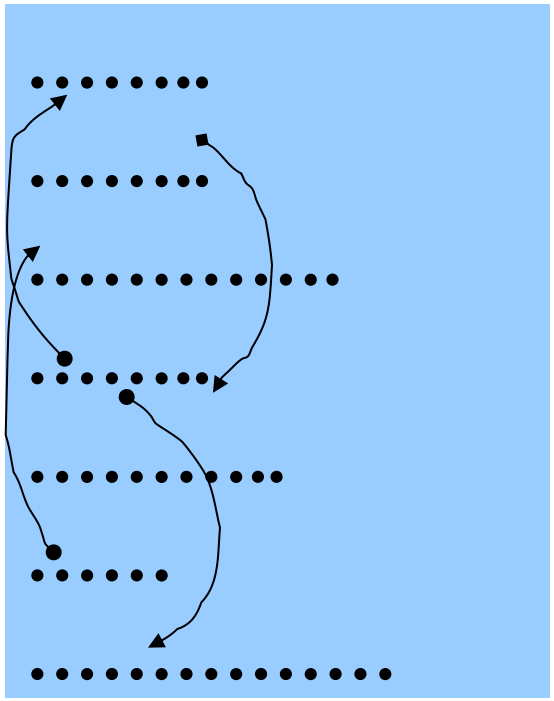
A chronology of selected programming languages

6.2 Traditional Programming Concepts

- High-level languages (C, C++, Java, C#, FORTRAN) include many kinds of abstractions
  - Simple: constants, literals, variables
  - Complex: statements, expressions, control
  - Esoteric: procedures, modules, libraries

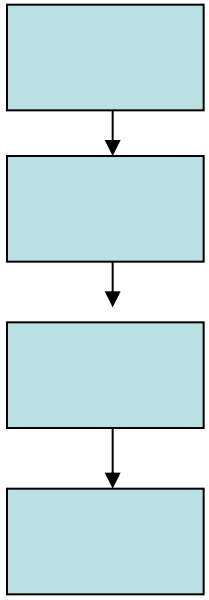
世博会远大馆 2 0 0 0平米、6层楼的建筑，工人用 2 4 小时建成--

初期的程序设计



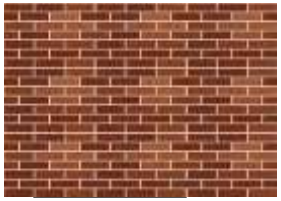
一碗面条式程序（BS）

结构化程序设计

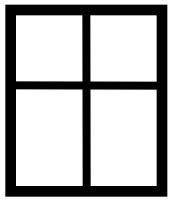


一串珠子式串连成

面向对象程序设计



拼装



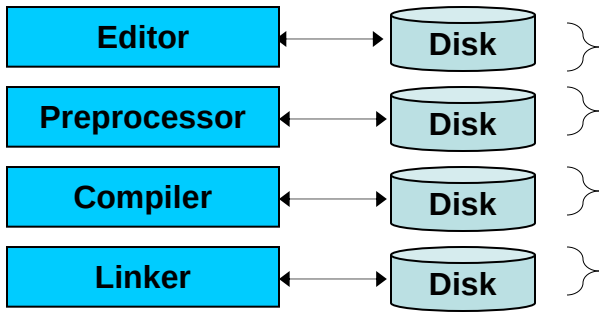
搭积木式

What Do They Have in Common?

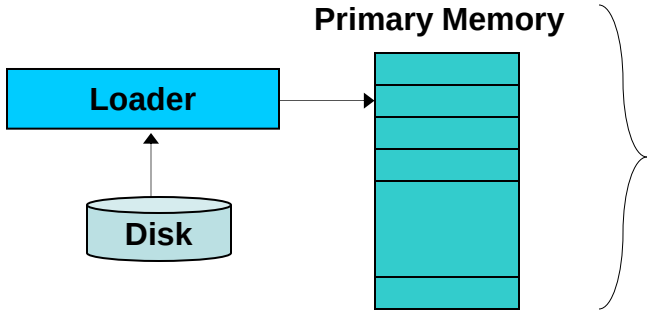
- Lexical structure and analysis词法结构分析
  - Tokens: keywords, operators, symbols, variables
  - Regular expressions and finite automata
- Syntactic structure and analysis语法结构分析
  - Parsing, context-free grammars
- Pragmatic issues语用问题
  - Scoping, block structure, local variables
  - Procedures, parameter passing, iteration, recursion
  - Type checking, data structures
- Semantics语义
  - What do programs mean and are they correct

A Typical C Program Development Environment

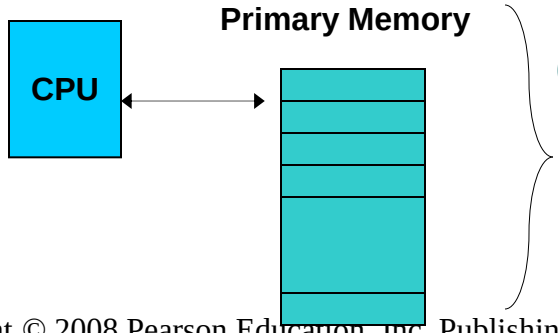
- **Phases of C Programs:**



1. Program is created in the editor and stored on disk
2. Preprocessor program processes the code
3. Compiler creates object code and stores it on disk.
4. Linker links the object code with the libraries



5. Loader puts program in memory.



6. CPU takes each instruction and executes it, possibly storing new data values as the program executes

1. *Edit*
2. *Preprocess*
3. *Compile*
4. *Link*

---

5. *Load*
6. *Execute*

Data Types

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

Variables and Data types

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

int   WeightLimit = 100;
```

Data Structure

- Conceptual shape or arrangement of data
- A common data structure is the **array**
  - In C

```
int Scores[2][9];
```

  - In FORTRAN

```
INTEGER Scores(2,9)
```

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

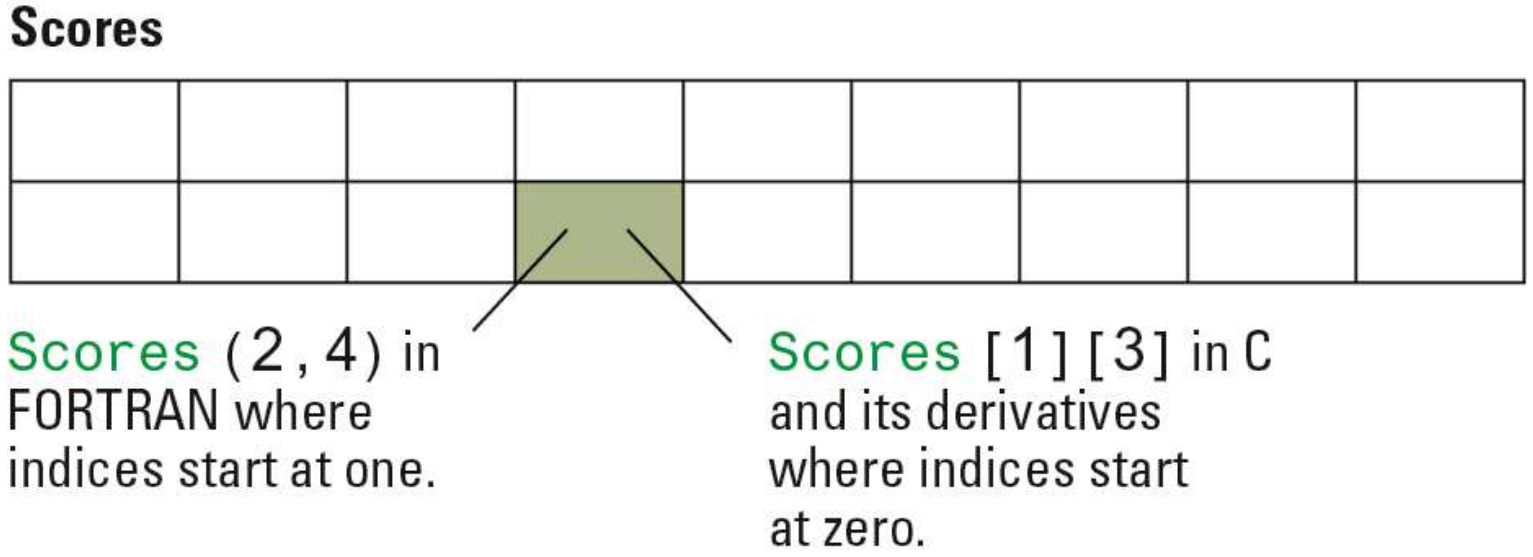
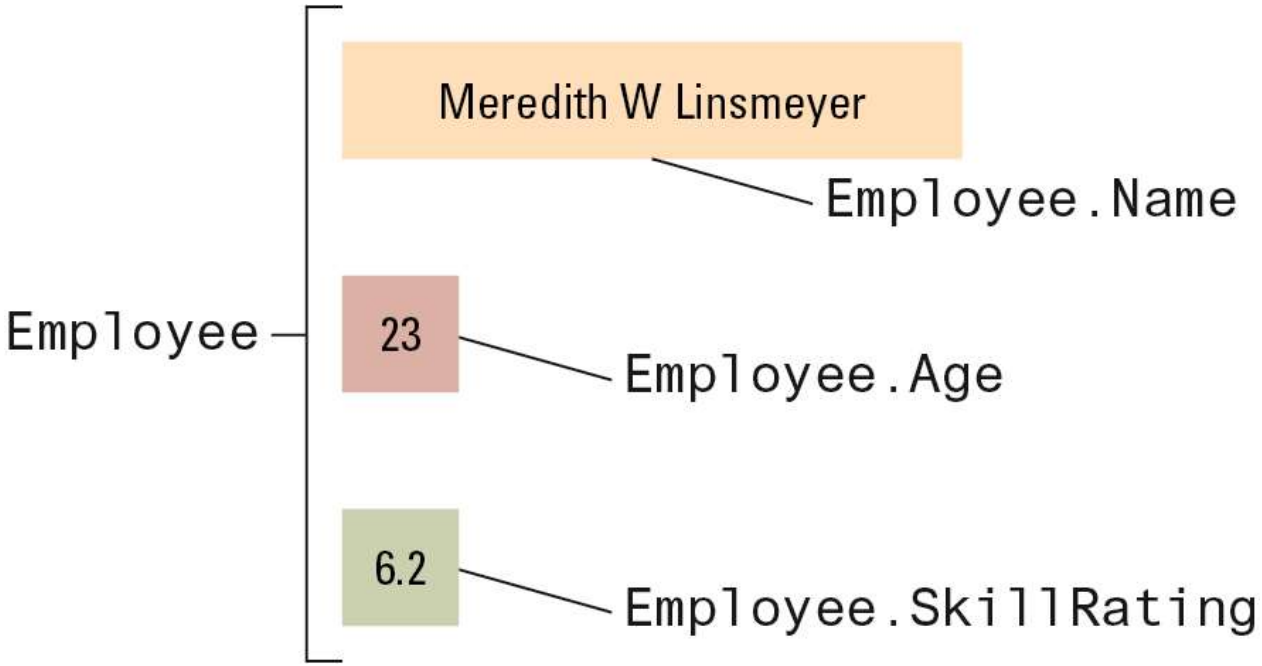


Figure 6.6 The conceptual structure of the aggregate type Employee




```
struct {    char  Name[25];
            int   Age;
            float SkillRating;
          } Employee;
```

Assignment Statements

- In C, C++, C#, Java

```
Z = X + y;
```

- In Ada

```
Z := X + y;
```

- In APL (A Programming Language)

```
Z ← X + y
```

Control Statements

- Go to statement

```
        goto 40
    20  Evade()
        goto 70
    40  if (KryptoniteLevel < LethalDose) then goto
60
        goto 20
    60  RescueDamsel()
    70  ...
```

- As a single statement

```
if (KryptoniteLevel < LethalDose):
    RescueDamsel()
else:
    Evade()
```

Control Statements (continued)

- If in Python

```
if (condition):
    statementA
else:
    statementB
```

- In C, C++, C#, and Java

```
if (condition) statementA; else statementB;
```

- In Ada

```
IF condition THEN
    statementA;
ELSE
    statementB;
END IF;
```

Control Statements (continued)

- While in Python

```
while (condition):
    body
```

- In C, C++, C#, and Java

```
while (condition)
{ body }
```

- In Ada

```
WHILE condition LOOP
    body
END LOOP;
```

Control Statements (continued)

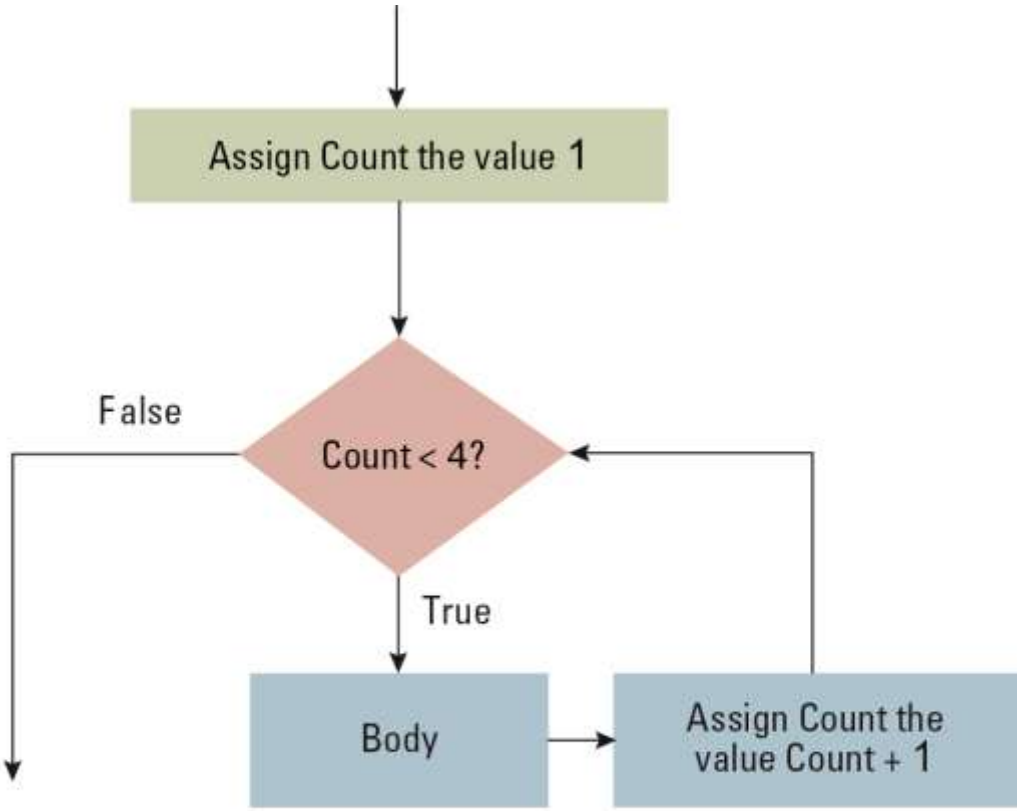
- Switch statement in C, C++, C#, and Java

```
switch (variable) {
    case 'A': statementA; break;
    case 'B': statementB; break;
    case 'C': statementC; break;
    default:  statementD; }
```

- In Ada

```
CASE variable IS
    WHEN 'A'=> statementA;
    WHEN 'B'=> statementB;
    WHEN 'C'=> statementC;
    WHEN OTHERS=> statementD;
END CASE;
```

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




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

Comments

- Explanatory statements within a program
- Helpful when a human reads a program
- Ignored by the compiler

```
/* This is a comment in C/C++/Java. */

// This is a comment in C/C++/Java.
```

6.3 Procedural Units

- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function
- Unit begins with the function's **header**
- Local versus Global Variables
- Formal Parameter (形参) and Actual Parameter (实参)
- Passing parameters by value versus reference

Figure 6.8 The flow of control involving a function

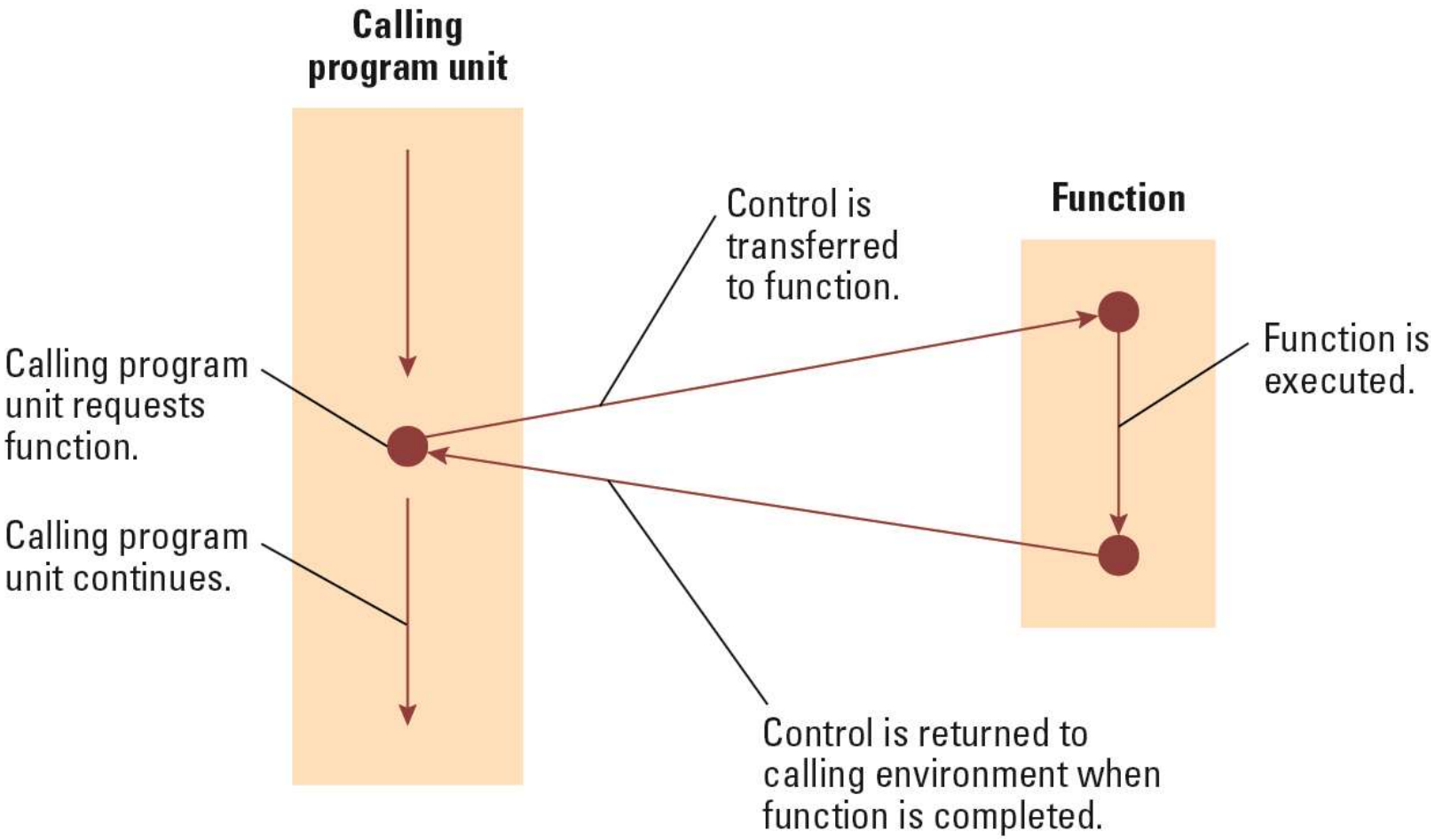


Figure 6.9 The function ProjectPopulation written in the programming language C

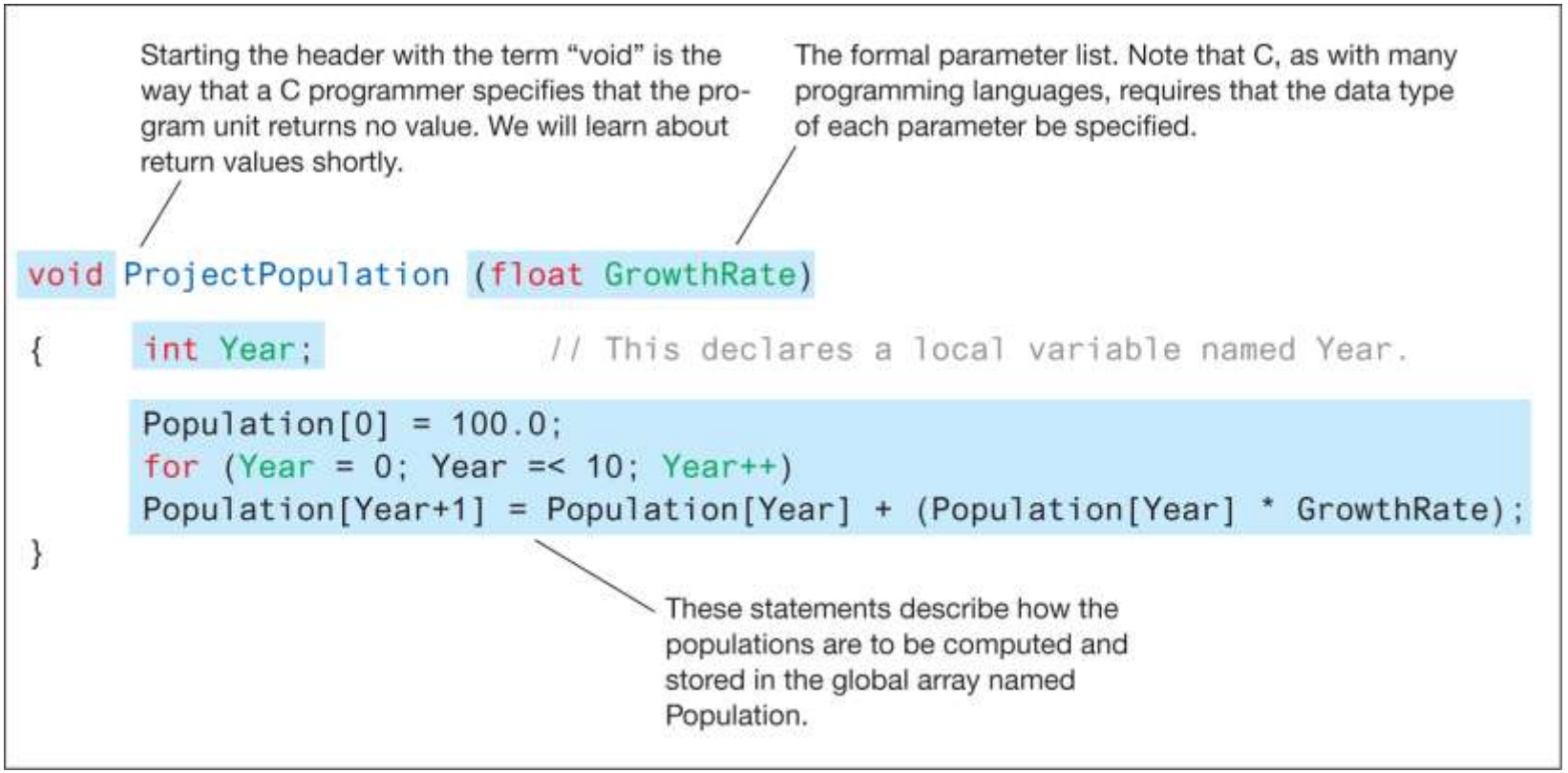


Figure 6.10 Executing the function Demo and passing parameters by value

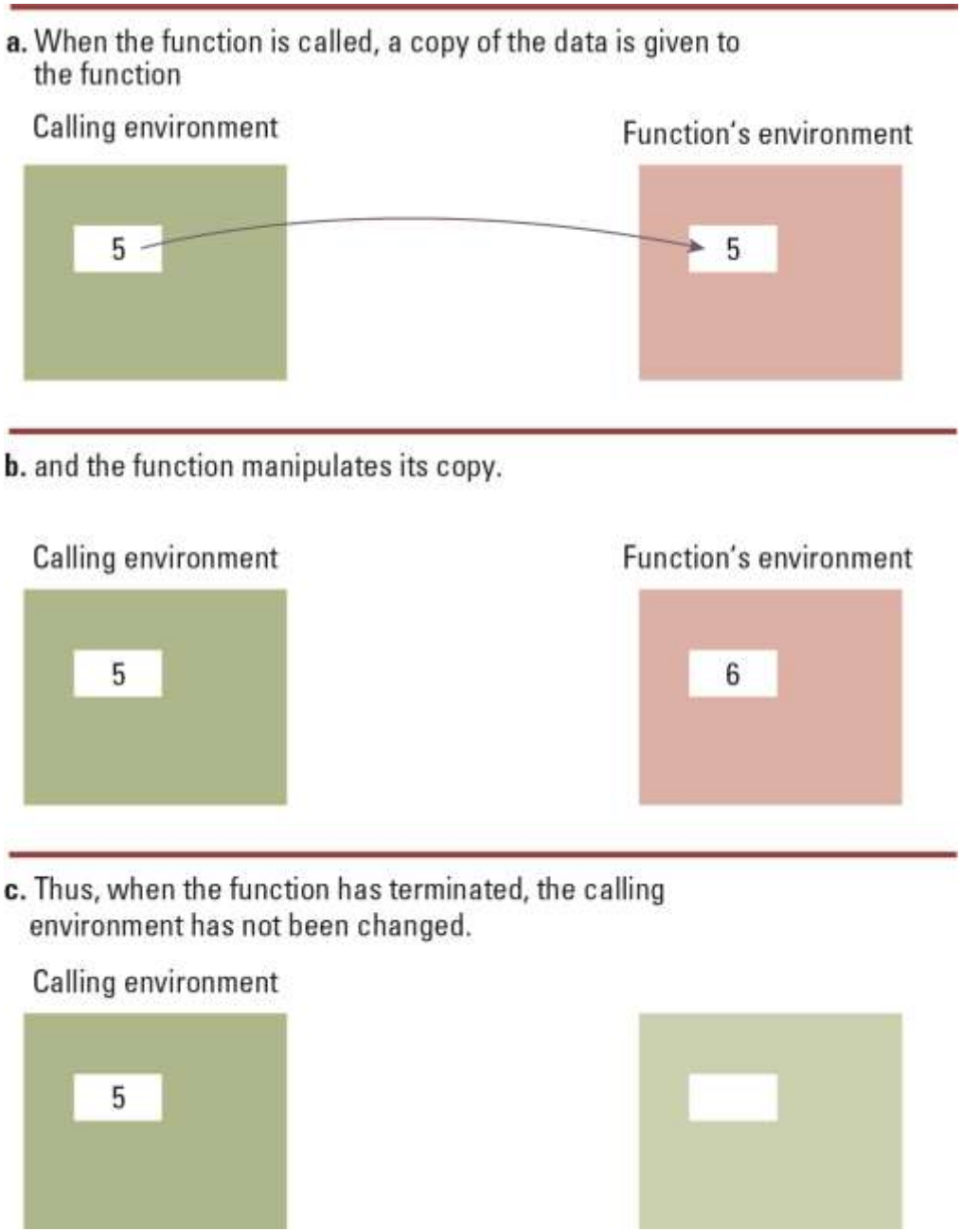


Figure 6.11 Executing the function Demo and passing parameters by reference

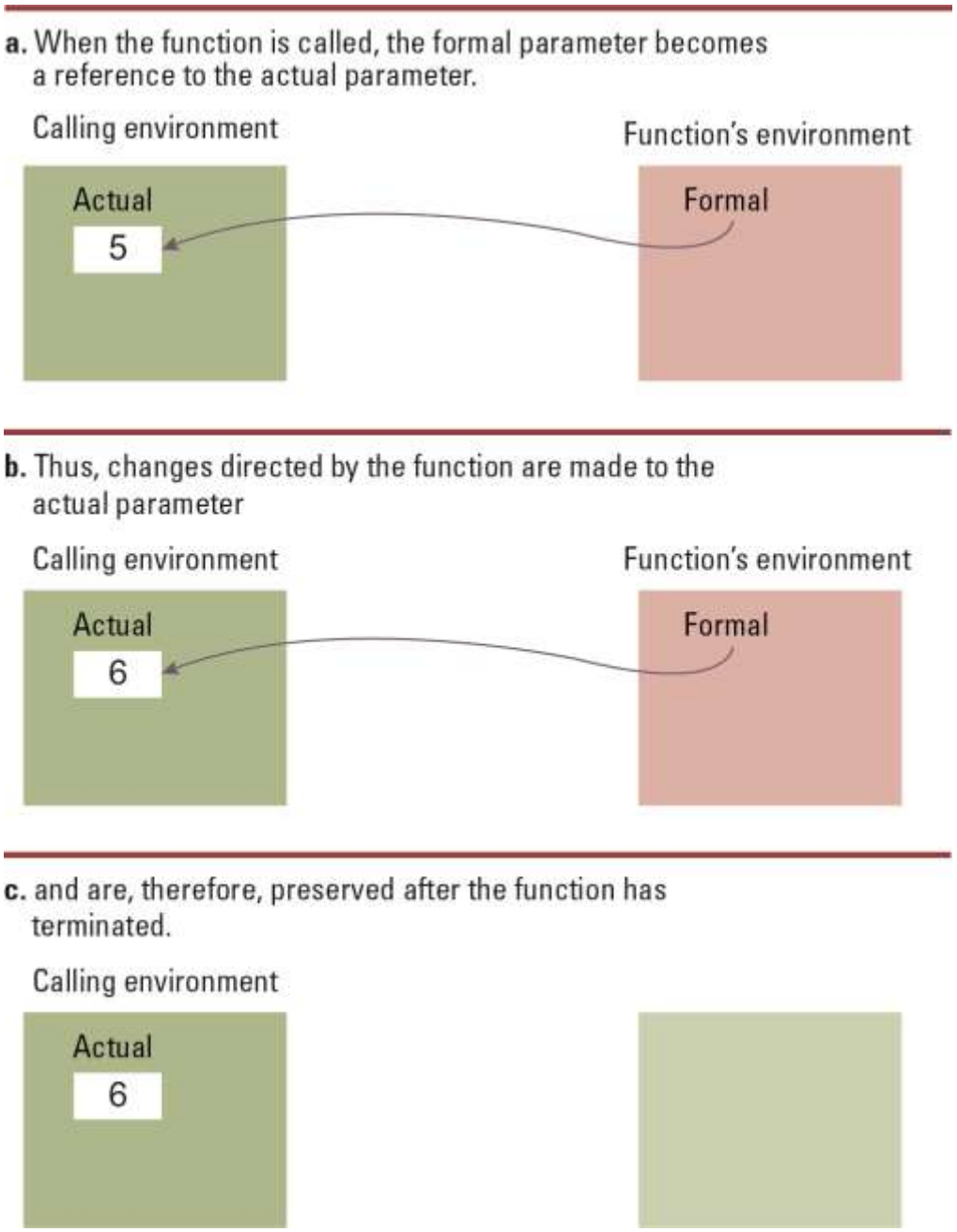
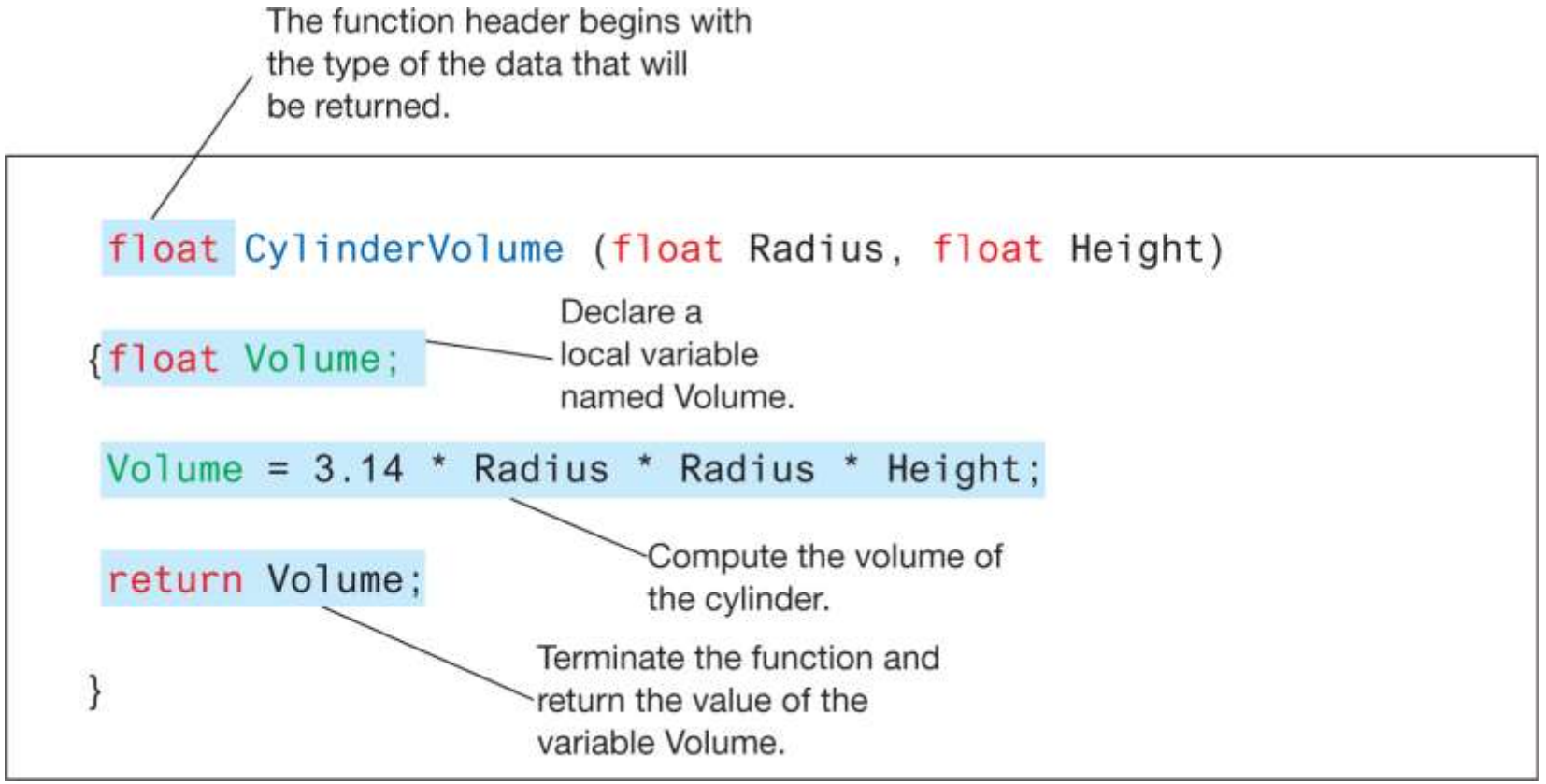


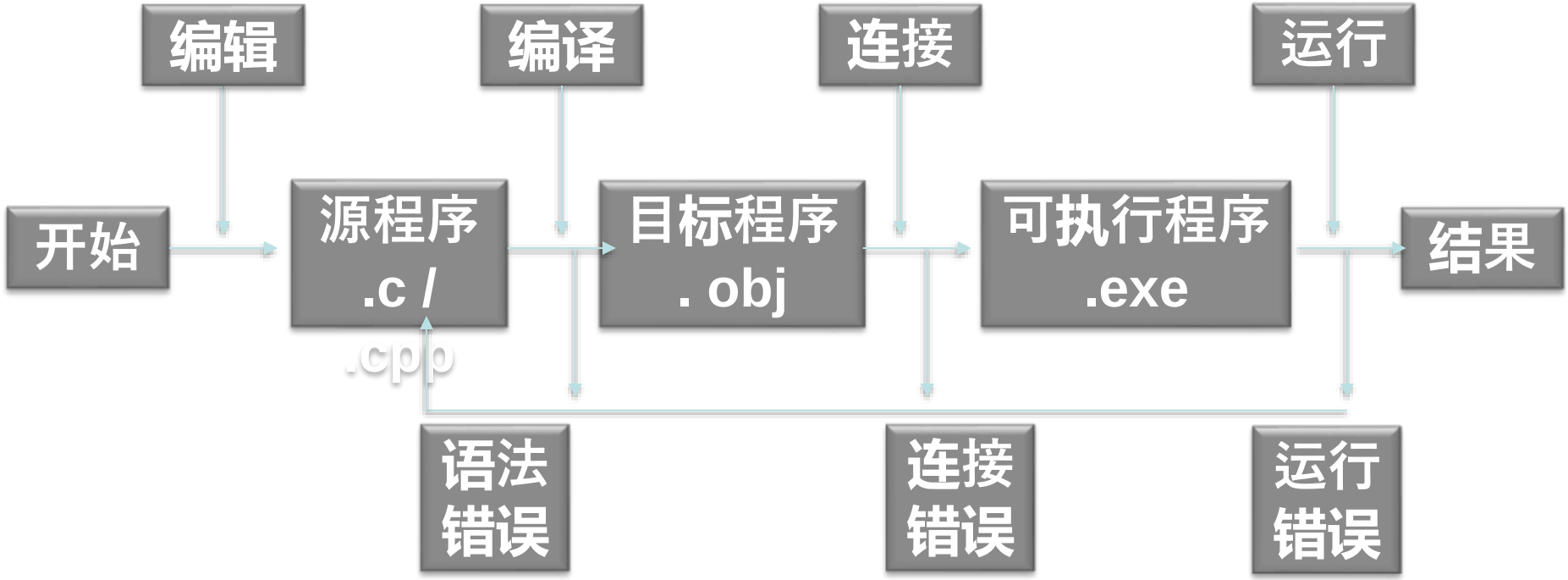
Figure 6.12 The fruitful function CylinderVolume written in the programming language C



Cost=CostPerVolUnit*CylinderVolume(3.35,12.7)

6.4 Language Implementation

程序的调试、运行步骤

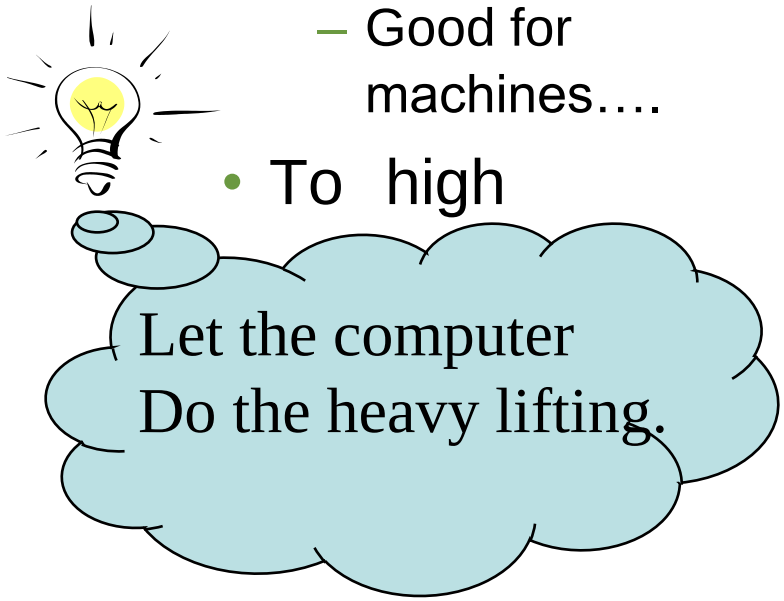


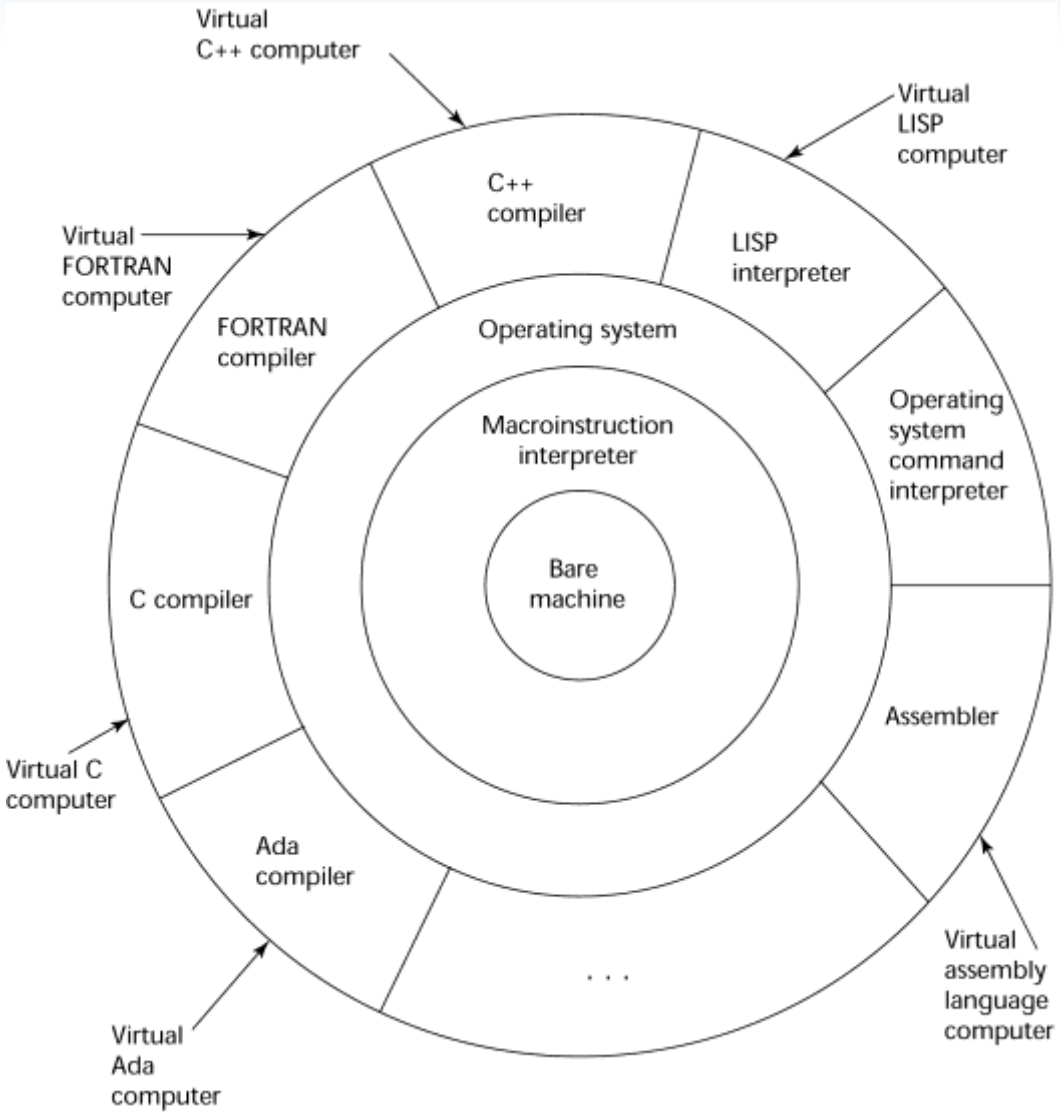
编译与解释

- 编译器是把源程序的每一条语句都编译成机器语言,并保存成二进制文件,这样运行时计算机可以直接以机器语言来运行此程序,速度很快 （如：C程序）
- 解释器则是只在执行程序时,才一条一条的解释成机器语言给计算机来执行,所以运行速度是不如编译后的程序运行的快的 （如：Python程序）

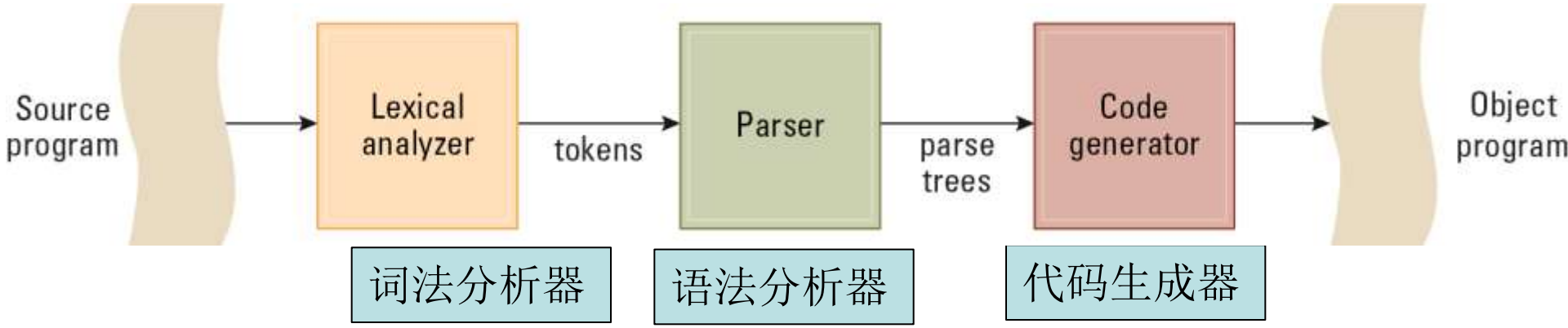
Translate… Why?

- Languages offer
  - Abstractions
  - At different levels
    - From low
      - Good for machines….
    - To high



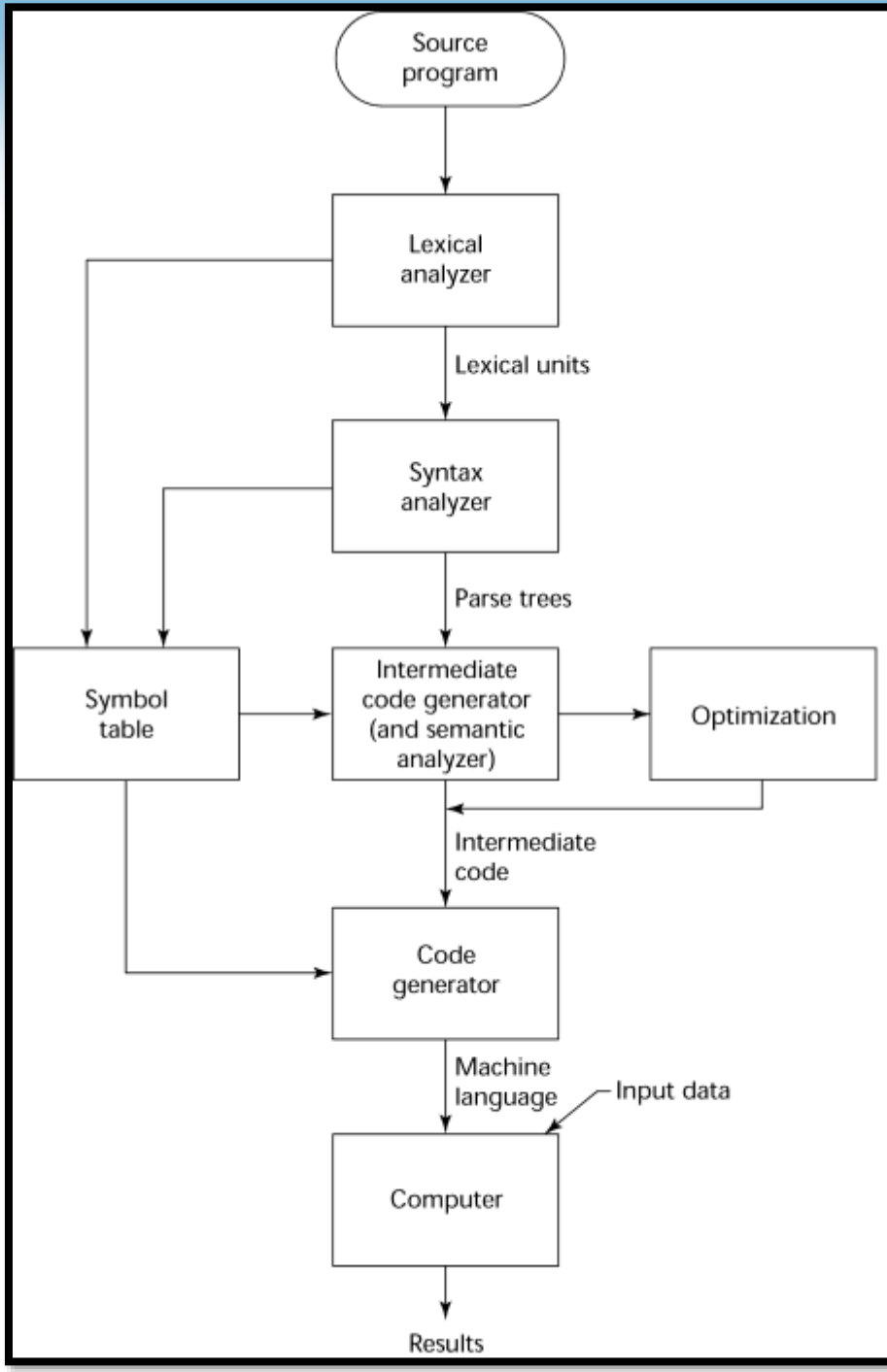


The translation process



Language Implementation

- The process of converting a program written in a high-level language into a machine-executable form.
  - The Lexical Analyzer recognizes which strings of symbols represent a single entity, or token.
  - The Parser groups tokens into statements, using syntax diagrams to make parse trees.
  - The Code Generator constructs machine-language instructions to implement the statements.



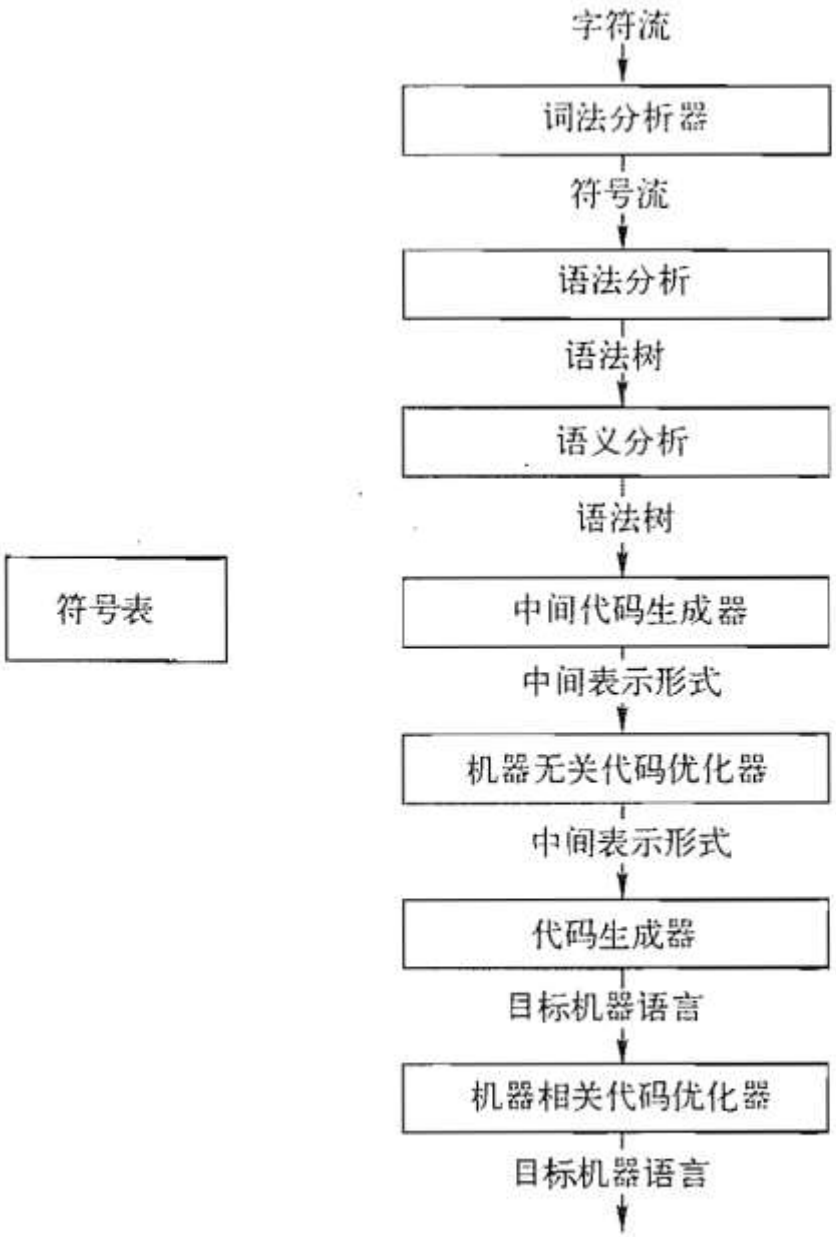


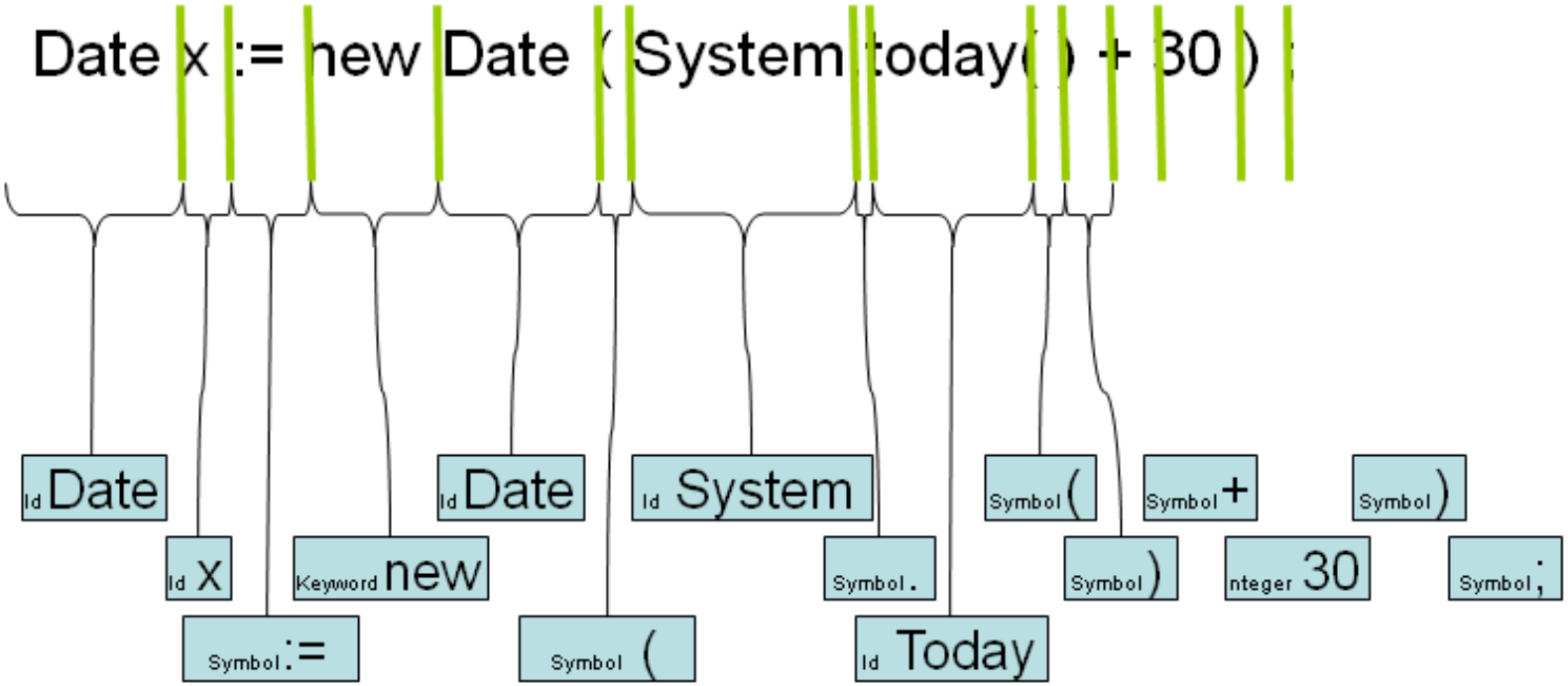
图 1-6　一个编译器的各个步骤

Major Phases of a Compiler

Major Phases of a Compiler

- Lexical analysis (词法分析)
  - Break the source into separate tokens

- **Lexical analysis**
  - *Slice* the sequence of *symbols* into *tokens*



- 词法分析
  - 编译器的第一个步骤称为词法分析或扫描。词法分析器读入组成源程序的字符流，并将其组成有意义的词素的序列。形如<token-name, attribute-value>这样的词法单元。（token-name是由语法分析使用的抽象符号，attribute-value是指向符号表中关于这个词法单元的条目，符号表条目的信息会被语义分析和代码生成步骤使用）

- 赋值语句：position = initial + rate * 60，对其进行词法分析得

抽象符号	词素
标识符 id	position
赋值运算符 =	=
标识符 id	initial
加法运算符 +	+
标识符 id	rate
乘法运算符 *	*
整数 60	60
空格（分析器直接忽略）	

- 经过词法分析之后，赋值语句的词法单元序列：<id, 1> < = > <id, 2> < + > <id, 3> < * > <60>