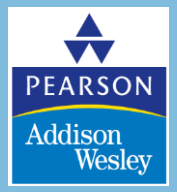
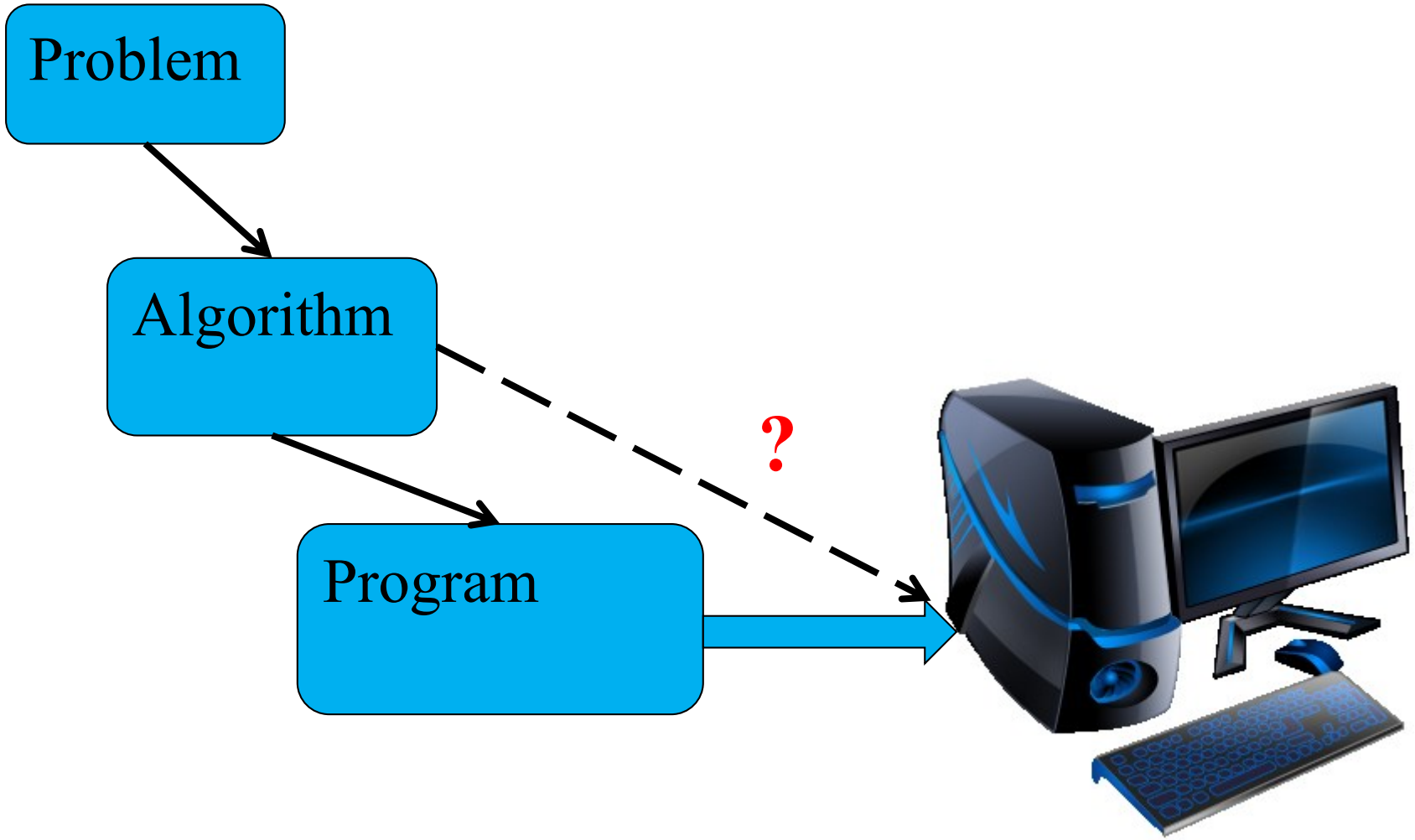
Chapter 6: Programming Languages

Computer Science: An Overview
Tenth Edition

by
J. Glenn Brookshear




Problem
Algorithm
Program
?

Question

- Source code to executable file?
  - Assembler
  - Interpreter
  - Compiler
  - Linker
  - Non of the above
  - All of the above

What is a Programming Language?

- A formal language for describing computation?
- A “user interface” to a computer?
- Syntax + semantics?
- Compiler, or interpreter, or “translator”?
- A tool to support a programming paradigm?

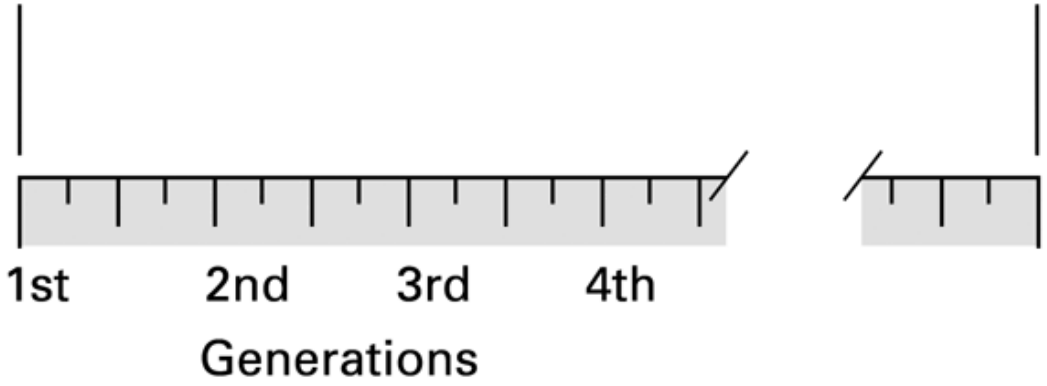
A programming language is a notational system for describing computation in a machine-readable and human-readable form.

— Louden

Programming Language generations

Problems solved in an environment in which the human must conform to the machine's characteristics

Problems solved in an environment in which the machine conforms to the human's characteristics



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 (op-rand)

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, ShippingCharge
5056	ADDI R0, R5 R6
30CE	ST R0, TotalCost
C000	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
```

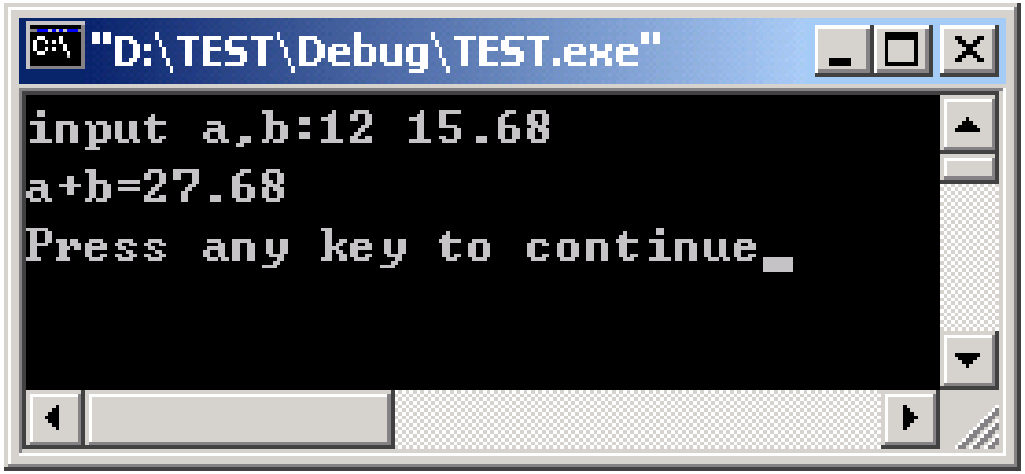
Object code

Third Generation Language

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

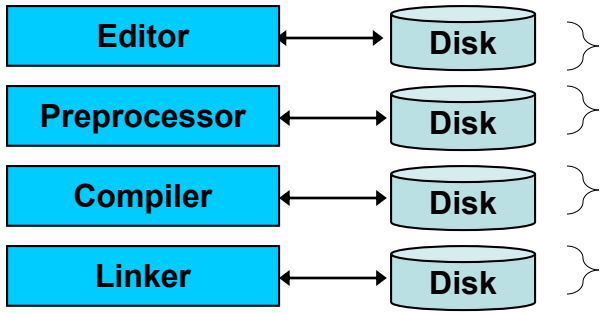
C program example

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

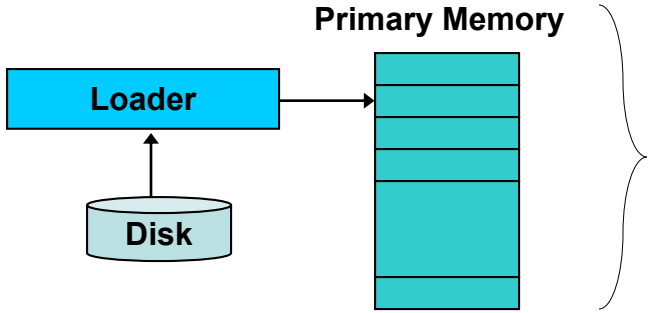


A Typical C Program Development Environment

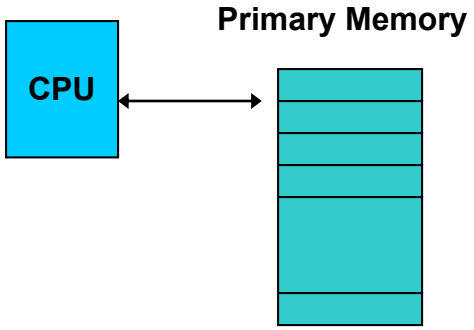
- **Phases of C Programs:**



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.



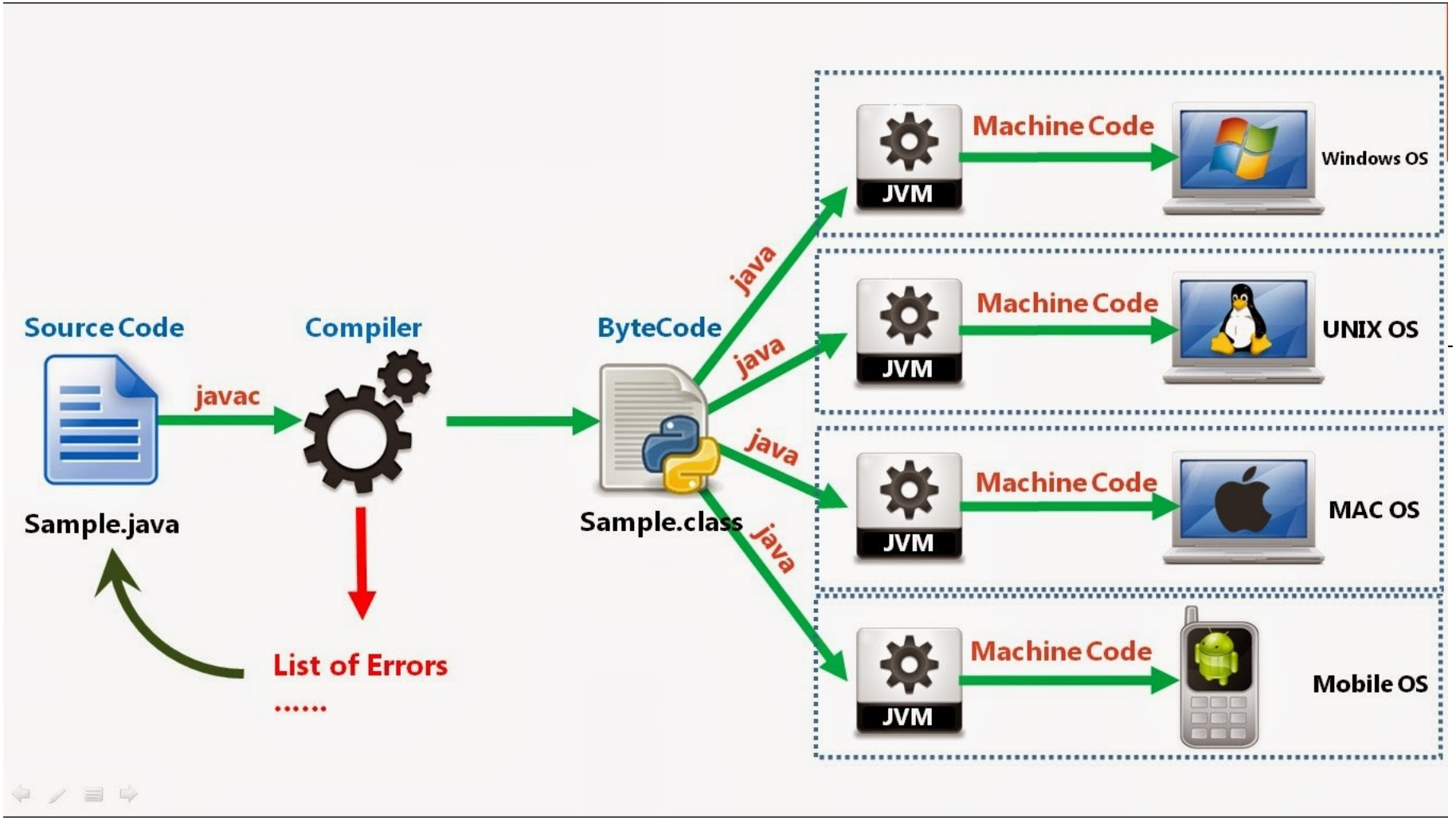
6. CPU takes each instruction and executes it, possibly storing new data values as the program executes

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

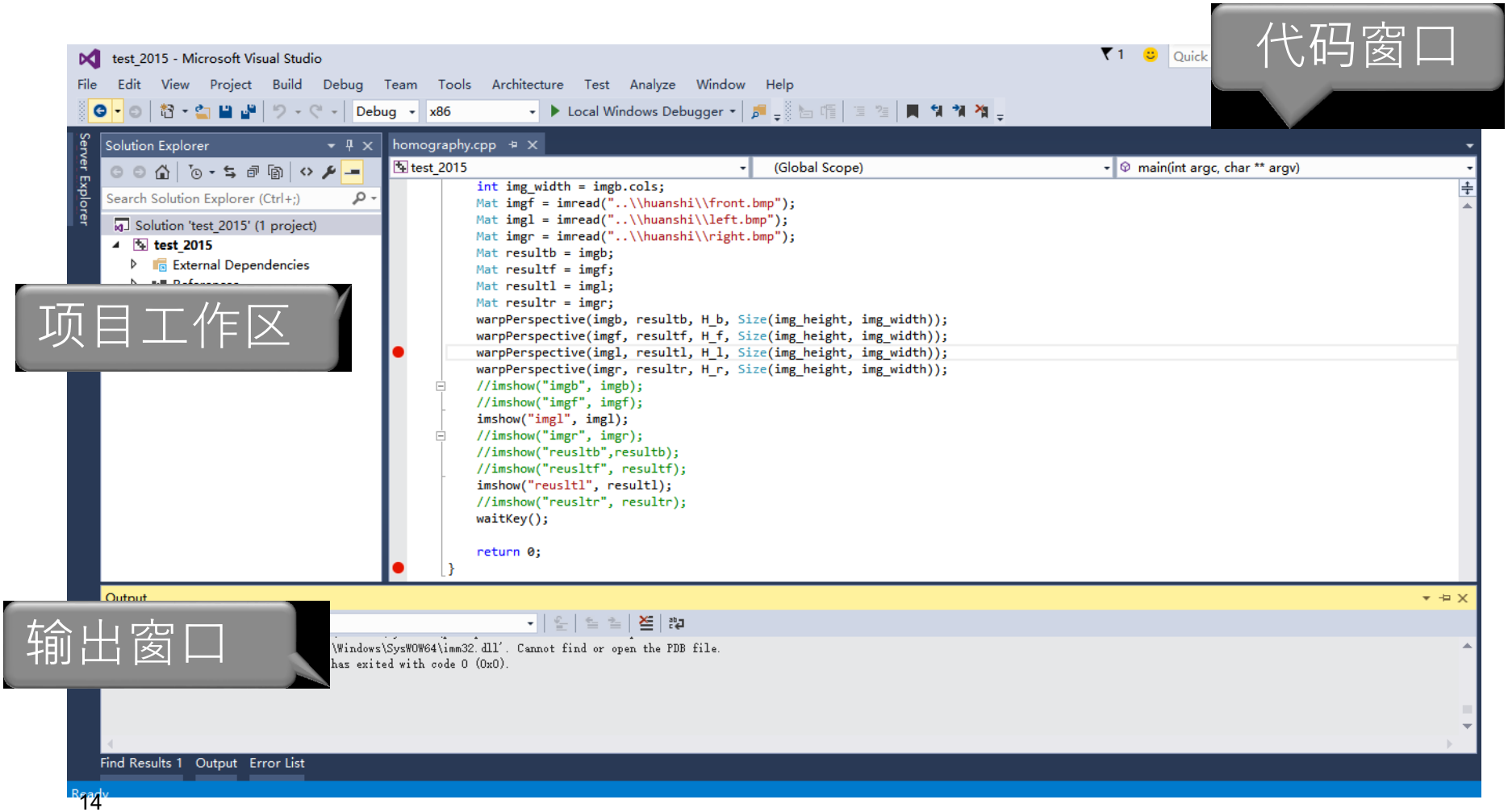
---

5. *Load*
6. *Execute*

JAVA


Source Code
Compiler
ByteCode
javac
Sample.java
Sample.class
List of Errors
.....
java
java
java
java
JVM
Machine Code
Windows OS
JVM
Machine Code
UNIX OS
JVM
Machine Code
MAC OS
JVM
Machine Code
Mobile OS

IDE: e.g. Visual Studio etc.



CMake

- https://cmake.org/
- CMakeLists.txt

```
john@TiEV:~$ cd Laser/build
john@TiEV:~/Laser/build$  cmake ..
john@TiEV:~/Laser/build$  make
john@TiEV:~/Laser/build$  ./Laser
```

```
cmake_minimum_required(VERSION 2.8)
project(Laser)

include_directories(/usr/local/include/)

set(SOURCE_FILES  main.cpp)

add_executable(Laser
            ${SOURCE_FILES})

target_link_libraries(Laser
       lcm
)
```

Makefile

- e.g.

```
subdirs:
        @list='$(SUBDIRS)'; \
        for subdir in $$list; do \
            echo $$subdir ; \
            $(MAKE) -C $$subdir || echo $(CURDIR)/$$subdir >>
$(ERRORLOG) ; \
        done
```

- 跟我一起写 Makefile
  http://blog.csdn.net/haoel/article/details/2886/

Compiler and Interpreter

No	Compiler	Interpreter
1	Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input .
2	Intermediate Object Code is **Generated**	**No** Intermediate Object Code is **Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement** : **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C Compiler	**Example** : BASIC

http://www.c4learn.com/c-programming/compiler-vs-interpreter/

Figure 6.2 The evolution of programming paradigms (编程范式)

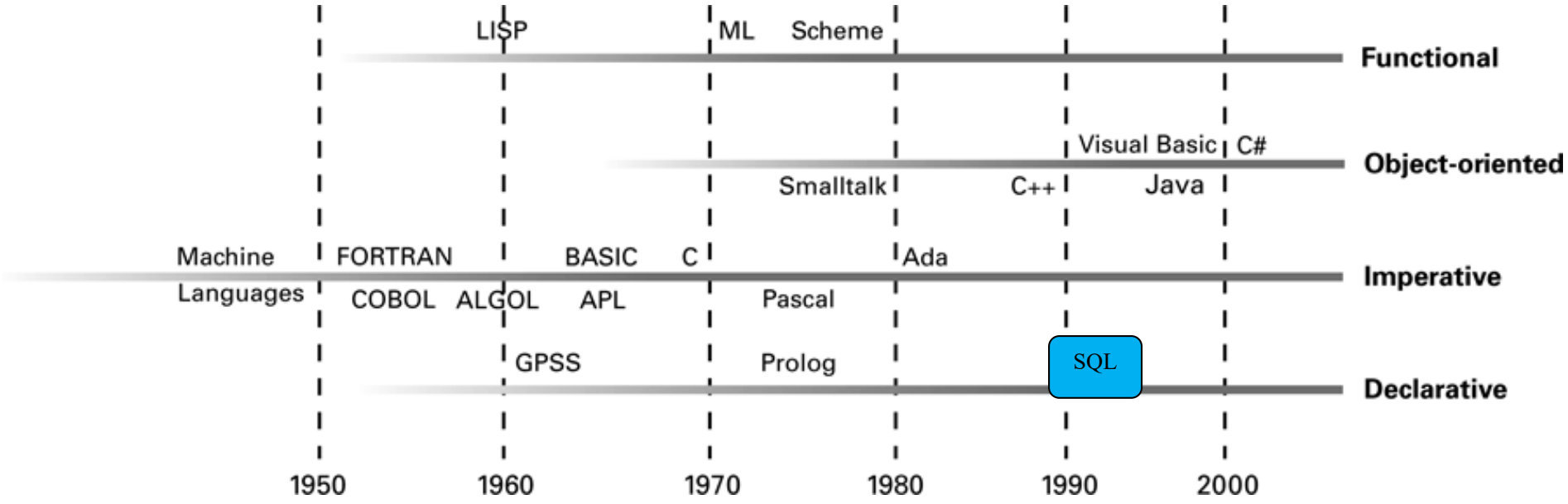
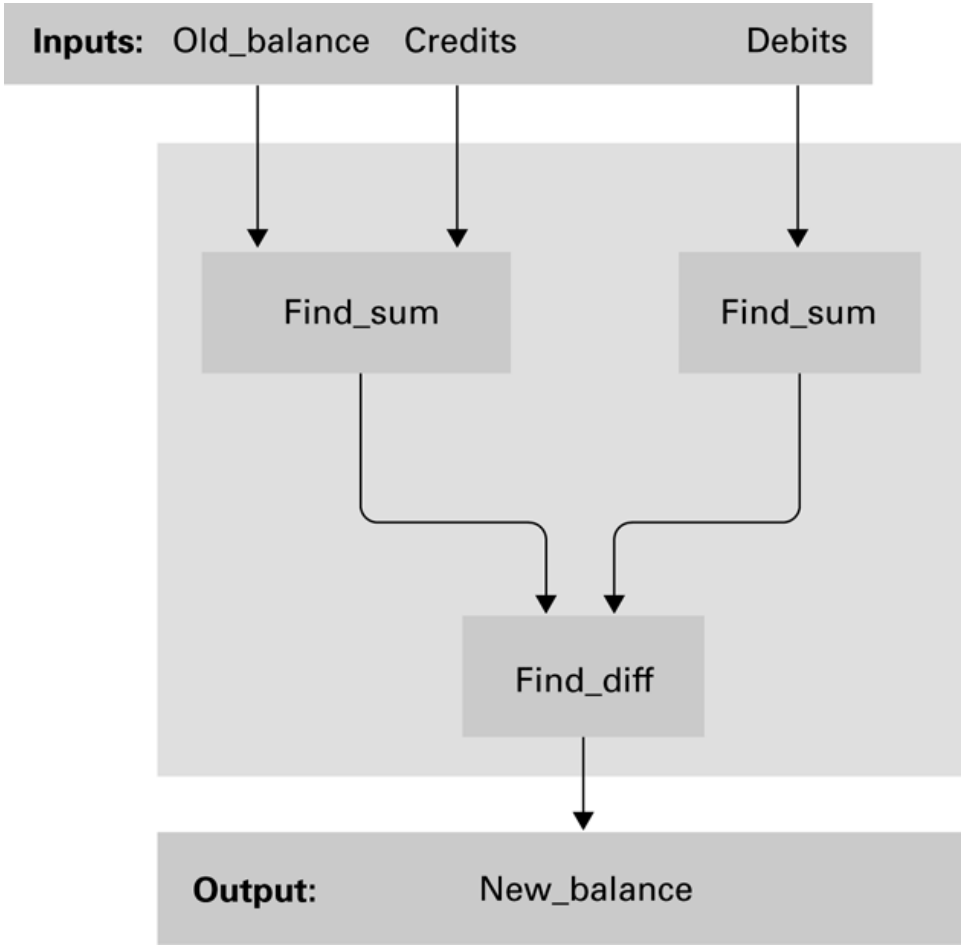


Figure 6.3 A function for checkbook balancing constructed from simpler functions



Compute a Fibonacci

Python

```
def fibonacci(n, first=0, second=1):
    while n != 0:
        print(first, second + "\n")
        n, first, second = n - 1, second, first + second # assignment
fibonacci(10)
```

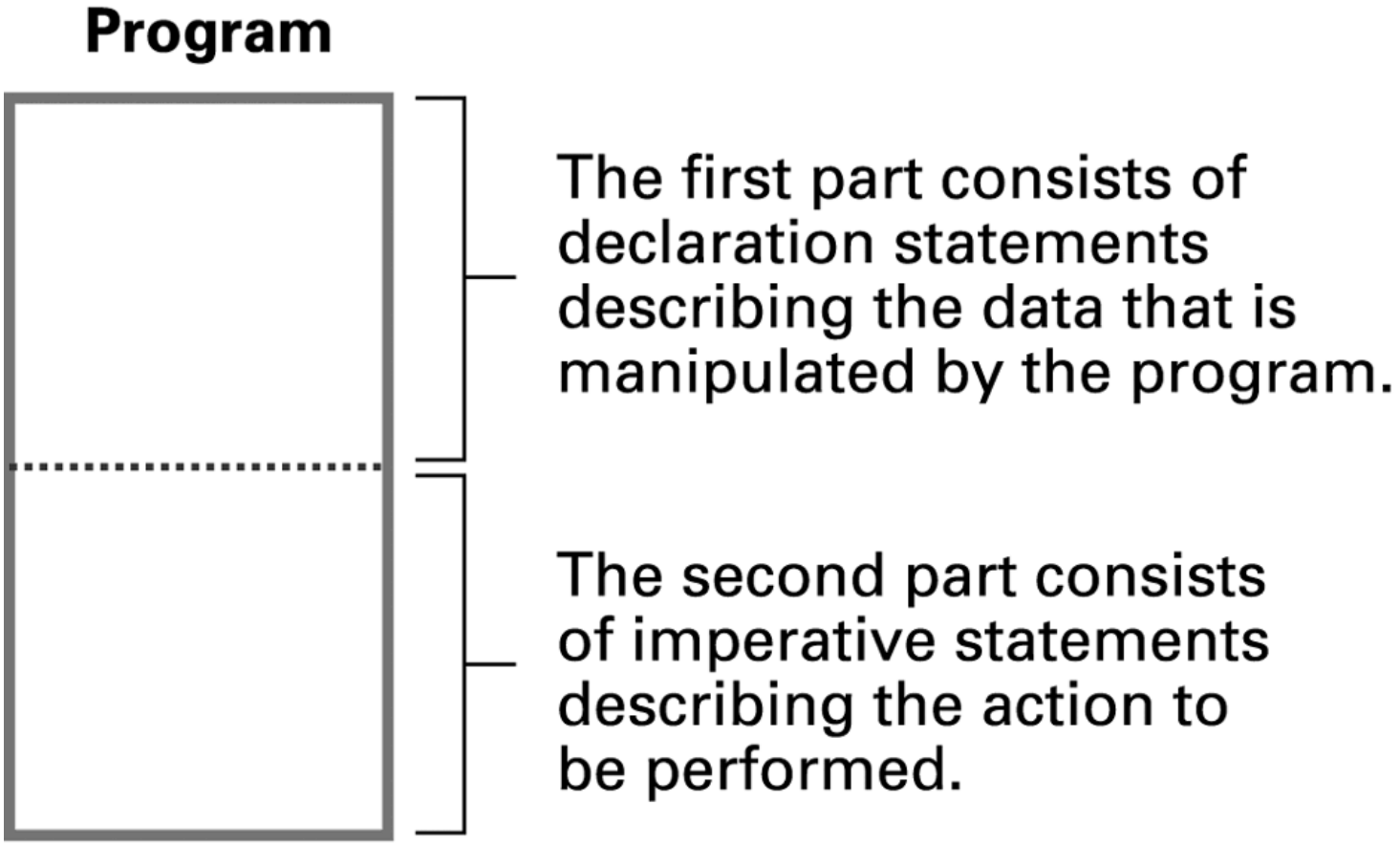
Python with Lambda (Functional)

```
fibonacci = (lambda n, first=0, second=1:
    "" if n == 0 else
    str(first) + "\n" + fibonacci(n - 1, second, first + second))
print(fibonacci(10))
```

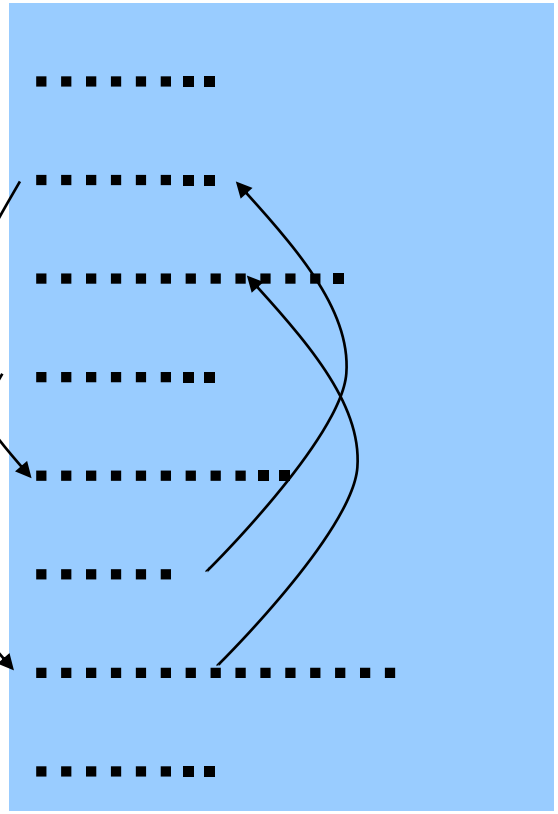
SQL Structured Query Language

- RDMS (Relational Database Management System)
- Declarative
- E.g.
  - SELECT * FROM Students;
  - SELECT Name, ID FROM Students;

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

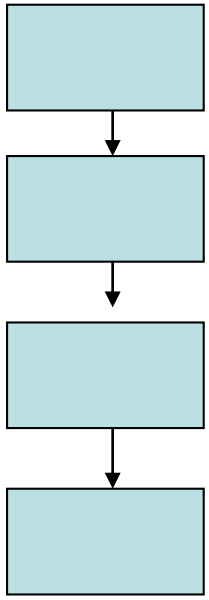


初期的程序设计



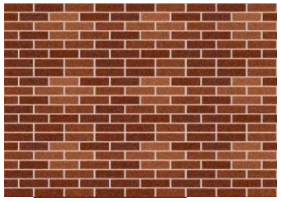
面条式程序

结构化程序设计

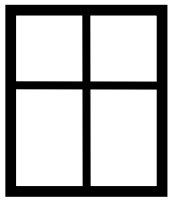


一串珠子式串连成

面向对象程序设计



拼装



搭积木式

常用高级语言 TIOBE Index

Nov 2018	Nov 2017	Change	Programming Language	Ratings	Change
1	1		Java	16.746%	+3.51%
2	2		C	14.396%	+5.10%
3	3		C++	8.282%	+2.94%
4	4		Python	7.683%	+3.20%
5	7		Visual Basic .NET	6.490%	+3.58%
6	5		C#	3.952%	+0.94%
7	6		JavaScript	2.655%	-0.32%
8	8		PHP	2.376%	+0.48%
9	-		SQL	1.844%	+1.84%
10	14		Go	1.495%	-0.07%
11	19		Objective-C	1.476%	+0.06%
12	20		Swift	1.455%	+0.07%
13	9		Delphi/Object Pascal	1.423%	-0.32%
14	11		R	1.407%	-0.20%
15	10		Assembly language	1.108%	-0.61%
16	13		Ruby	1.091%	-0.50%
17	12		MATLAB	1.030%	-0.57%
18	15		Perl	1.001%	-0.56%
19	18		PL/SQL	1.000%	-0.45%
20	17		Visual Basic	0.854%	-0.63%

2018.11

- Questions?

- What are the components of a programming language?
- Quick tour…

- Declarative statements
- Imperative statements
- Comments

Variable Declarations

```
float     Length, Width;
int       Price, Total, Tax;
char      Symbol;
vector<string>   name_list;
```

Data Types

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

Data structure

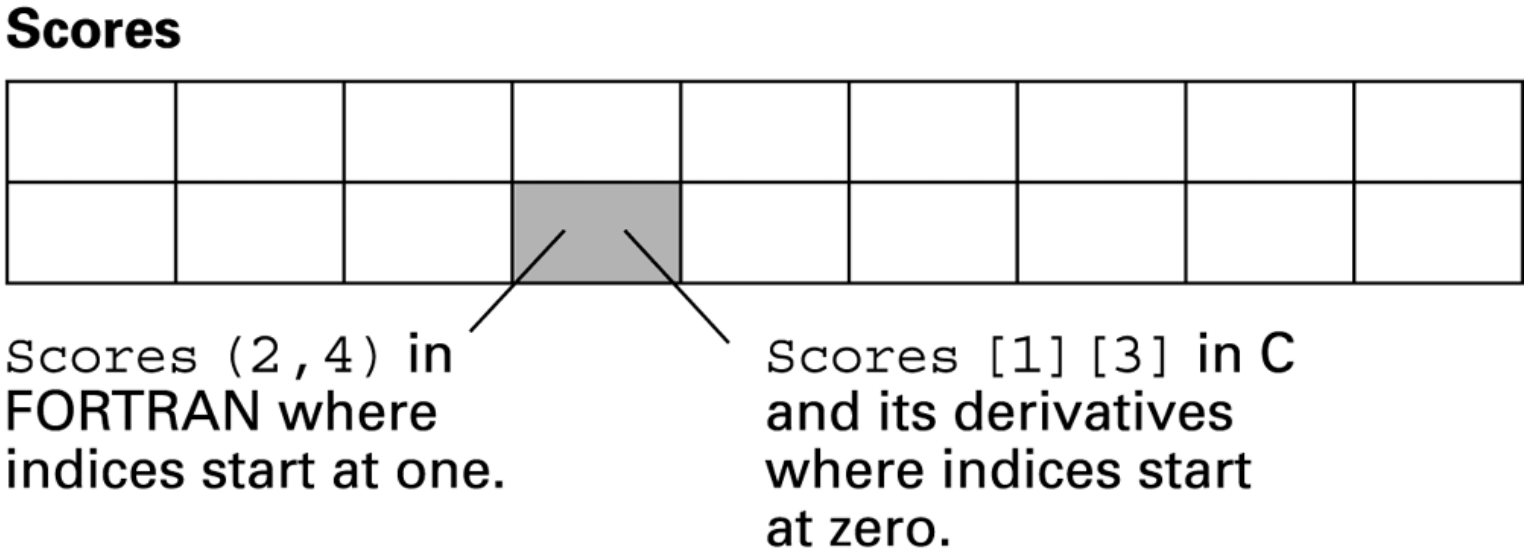
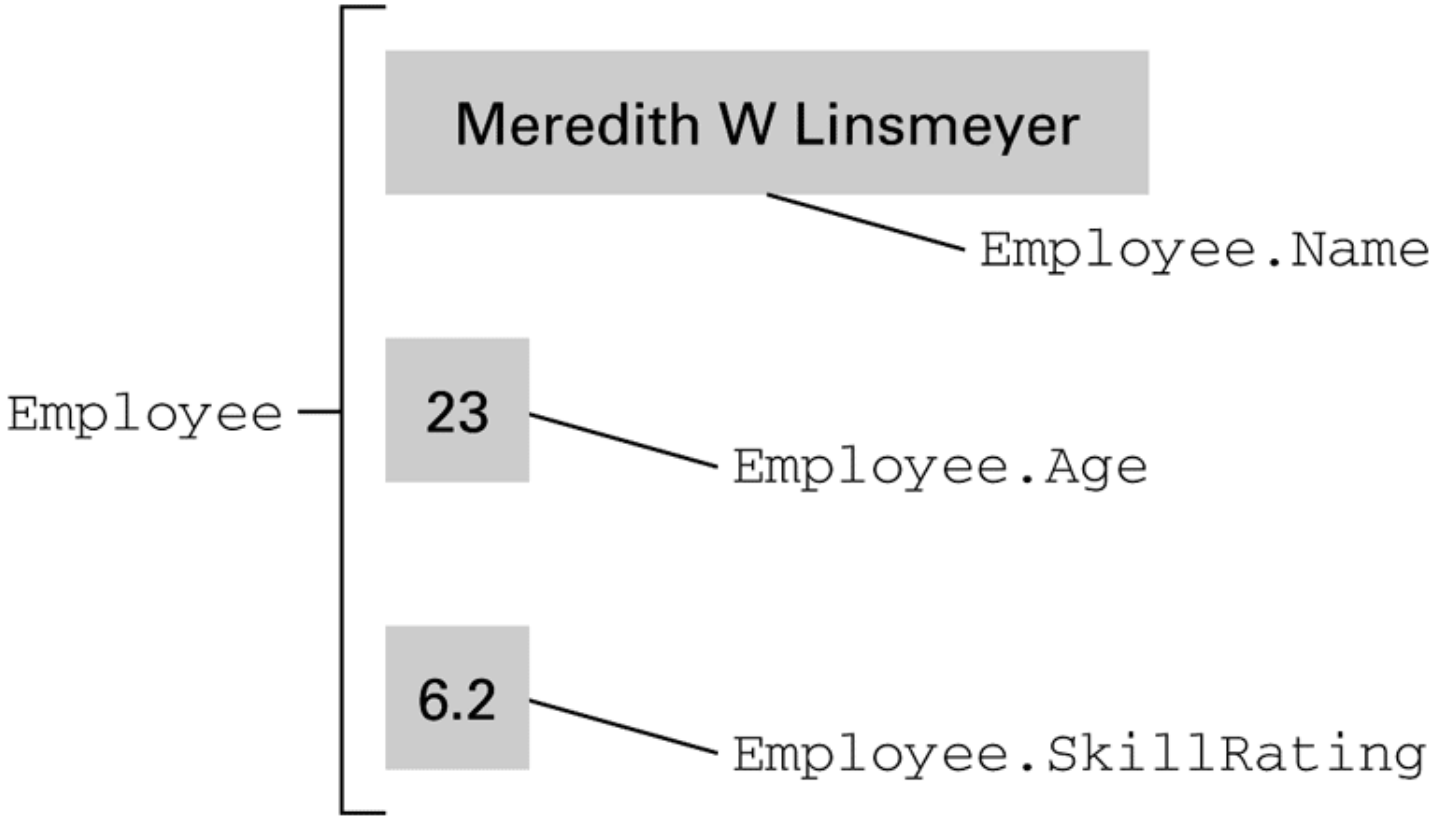


Figure 6.5 A two-dimensional array with two rows and nine columns

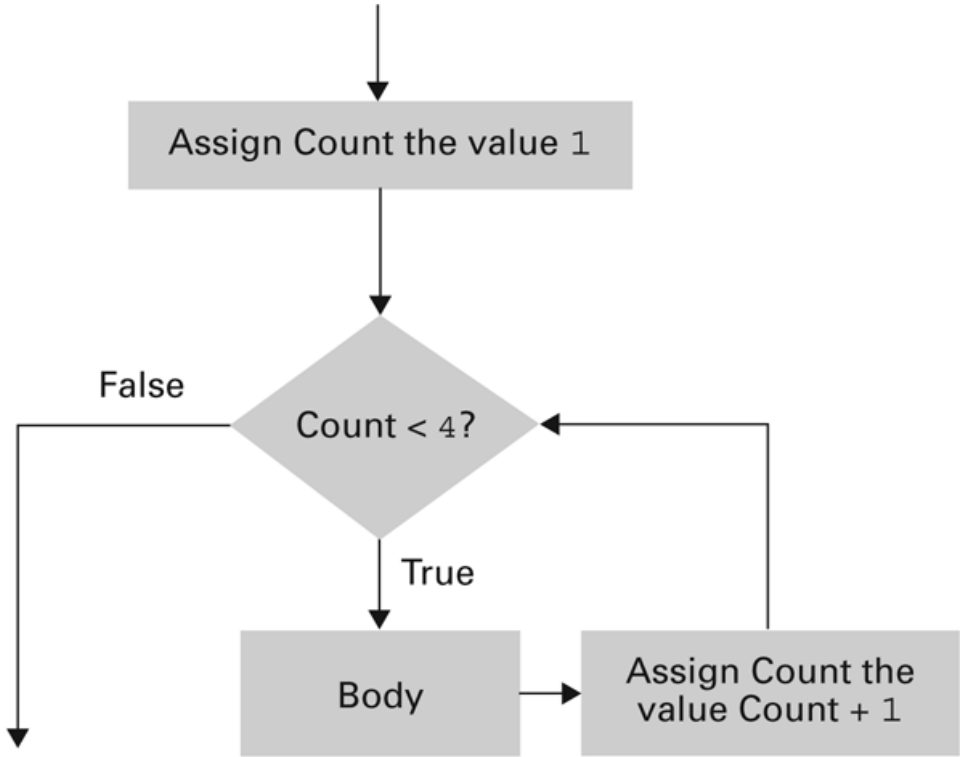
Figure 6.6 The conceptual structure of the heterogeneous array Employee



Constants and Literals

- Area = r*r*3.1415
- Const float Pi = 3.1415
  - Area = r*r*Pi

Control statements




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

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

Procedural Units

- Procedures versus Functions
- 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 procedure

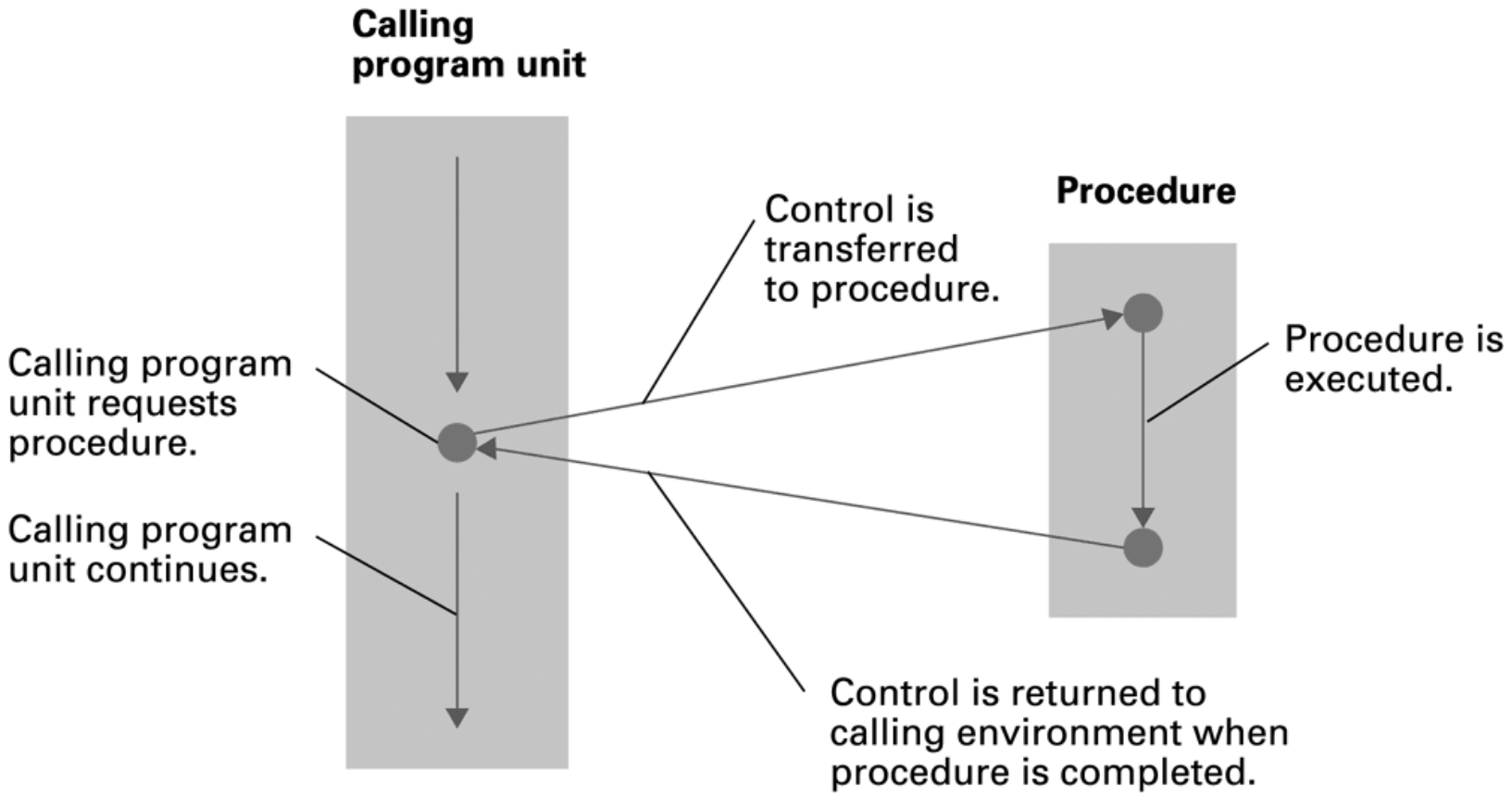


Figure 6.9 The procedure ProjectPopulation written in the programming language C

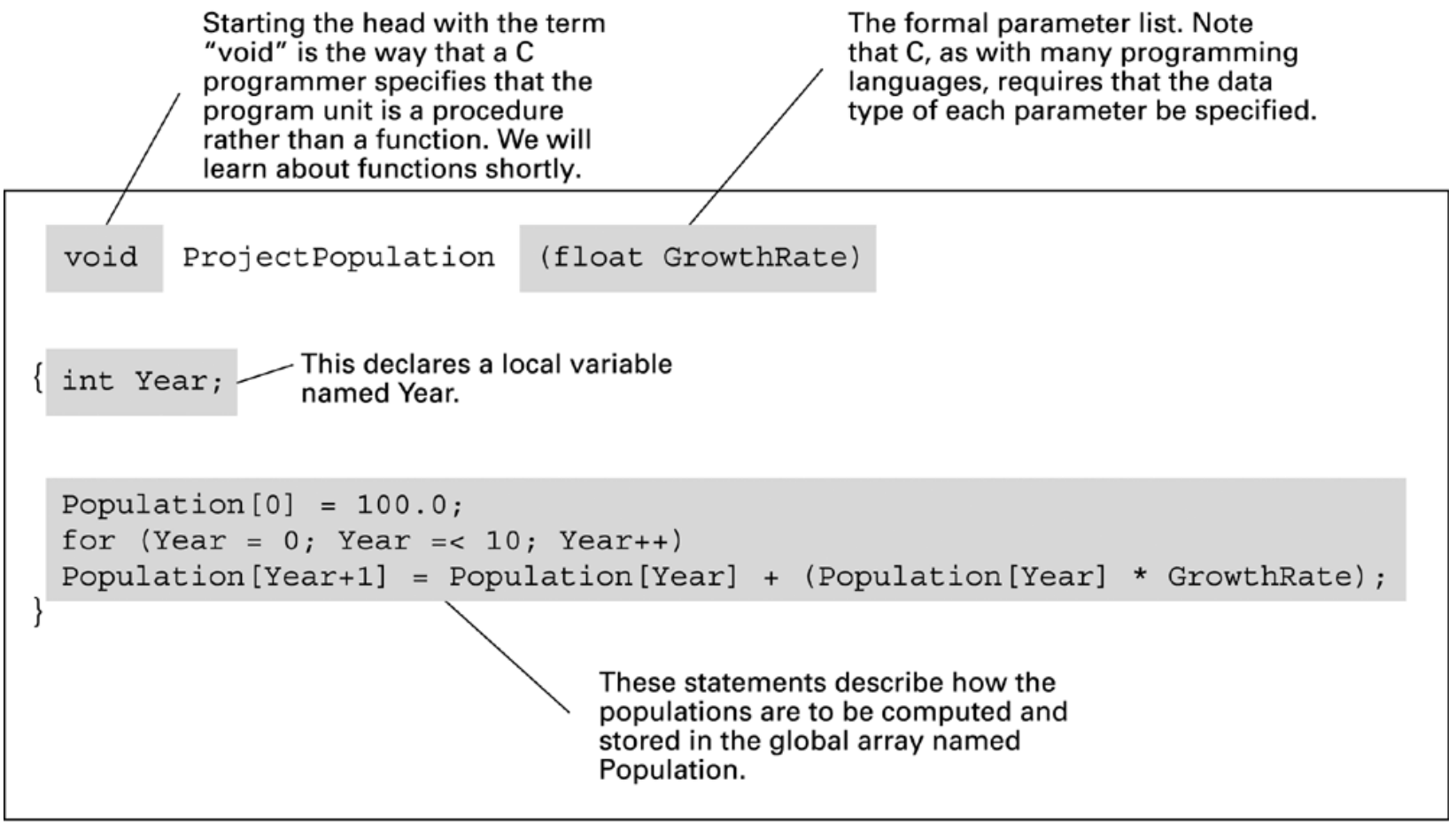


Figure 6.10 Executing the procedure Demo and passing parameters by value

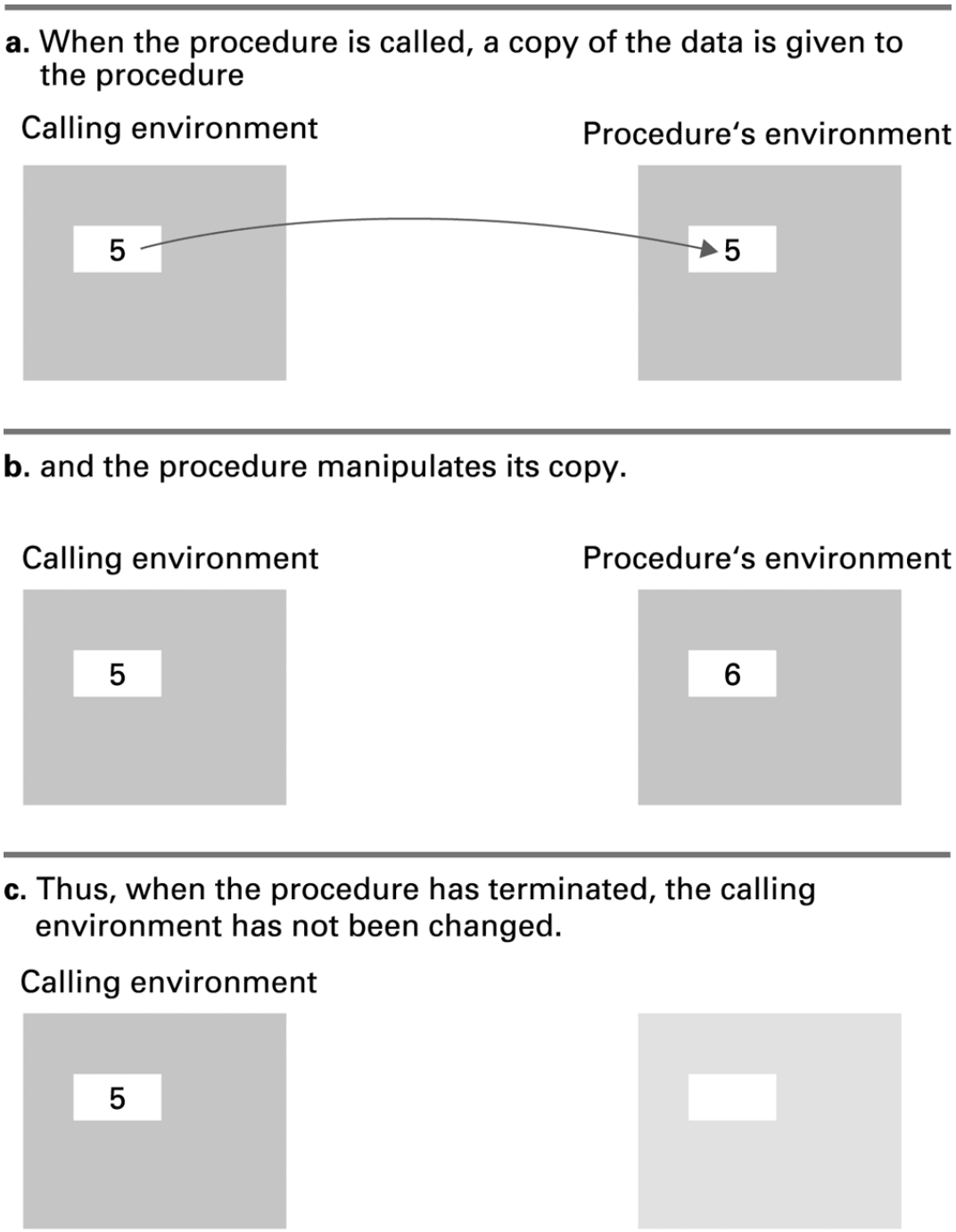


Figure 6.11 Executing the procedure Demo and passing parameters by reference

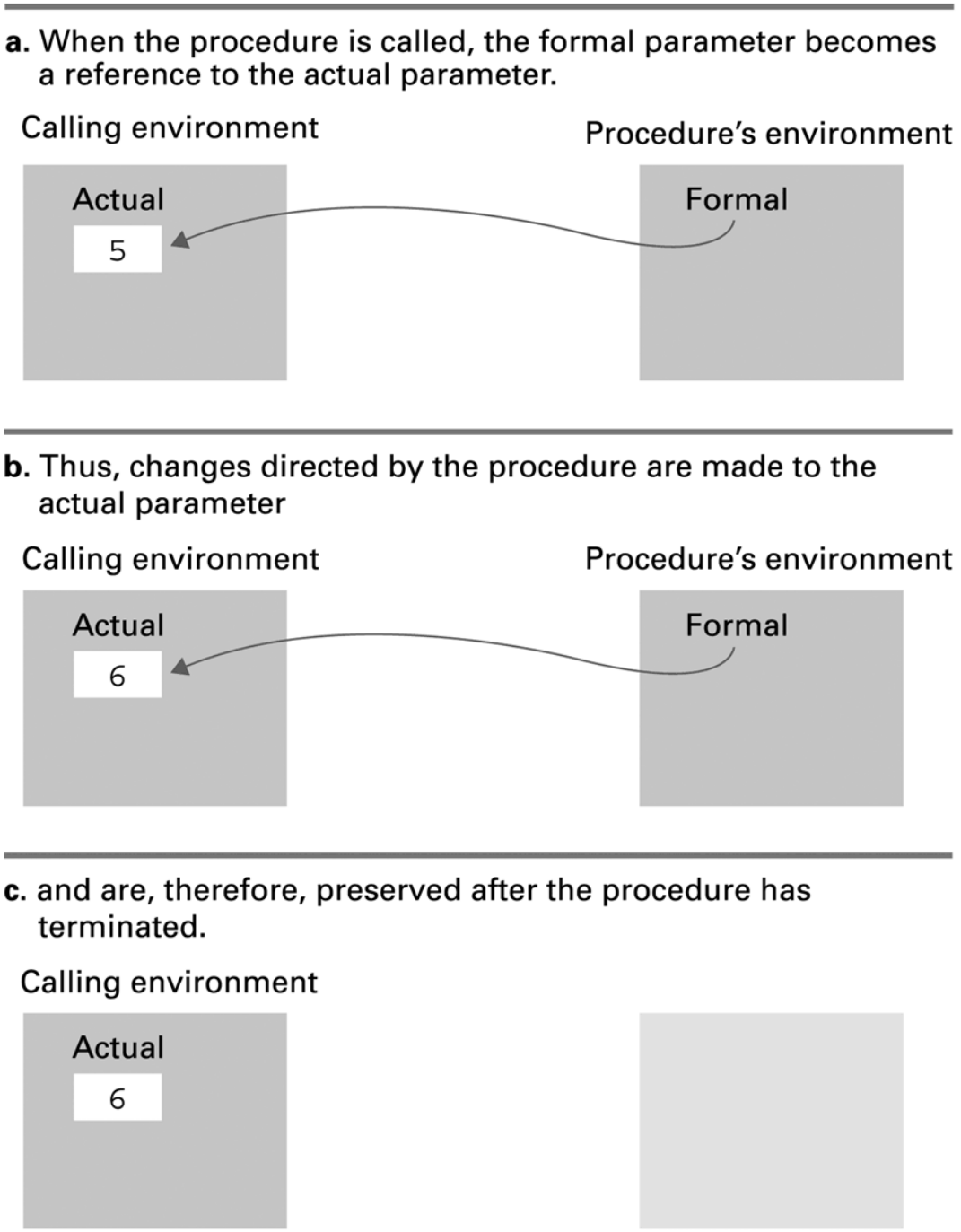
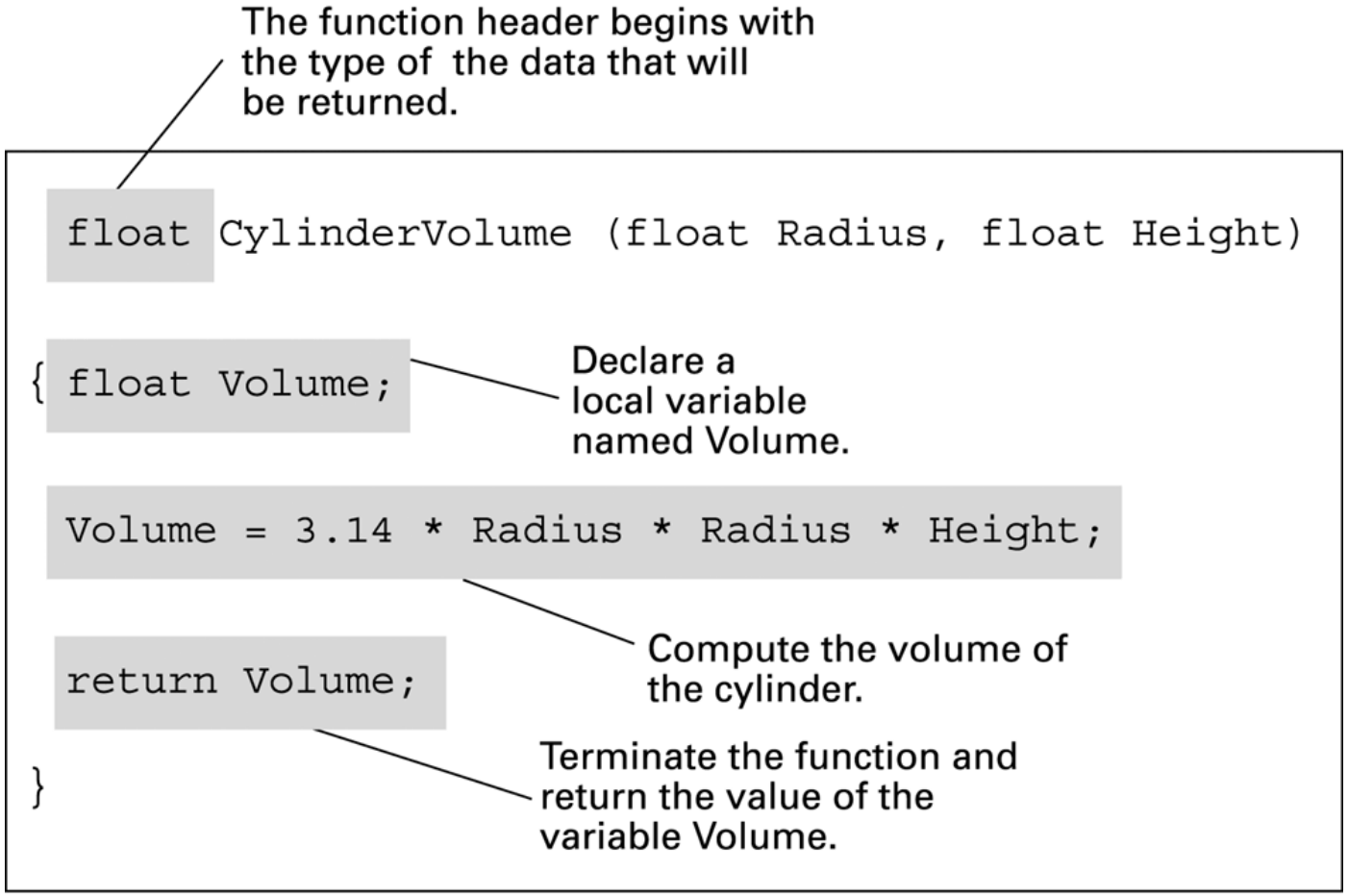


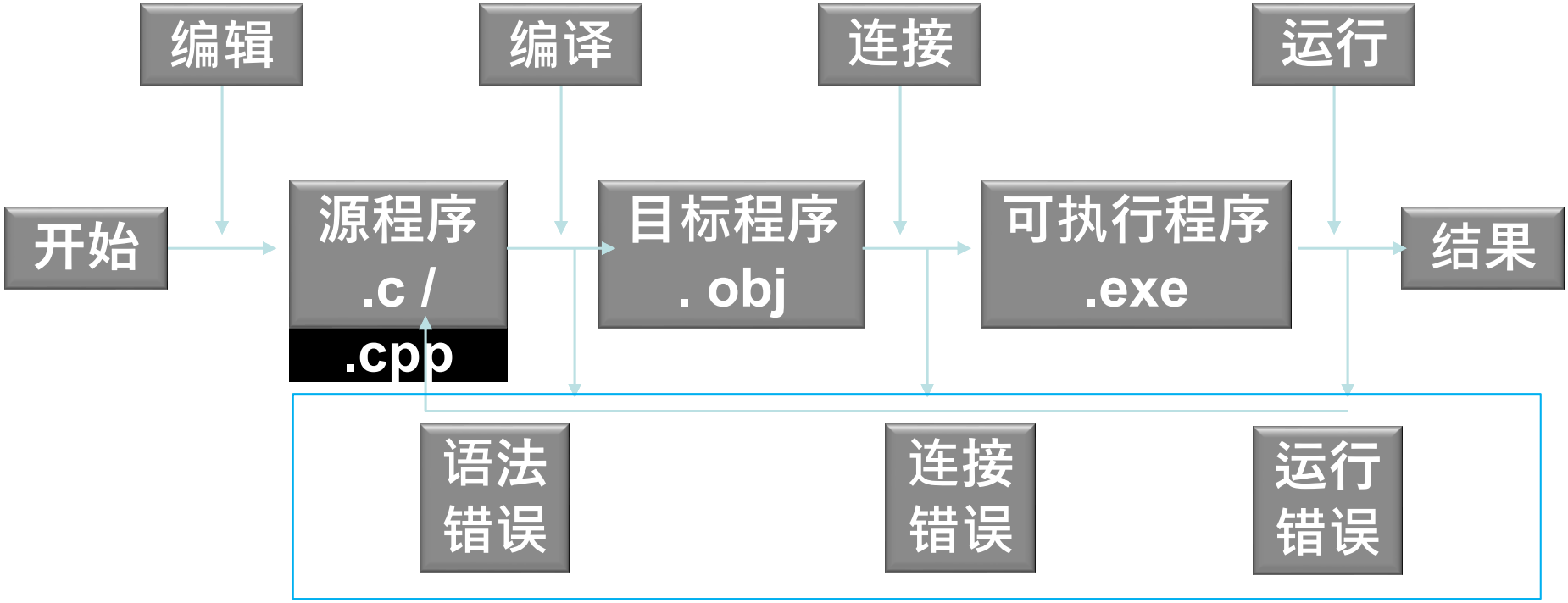
Figure 6.12 The function CylinderVolume written in the programming language C



Comments

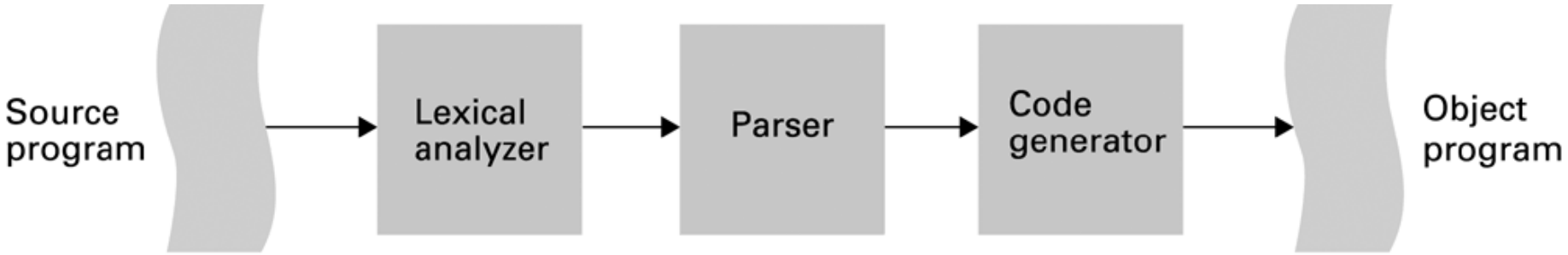
- //This is a comment
- /*This is a comment*/
- #This is a comment
- <!-- This is a comment-->
- --This is a comment
- %This is a comment%

Debug



- Let’s go a little deeper…

Figure 6.13 **The translation process**



Lexical analysis (词法分析)

```
sum = 3 + 2;
```

Lexeme	Token
sum	"Identifier"
=	"Assignment operator"
3	"Integer literal"
+	"Addition operator"
2	"Integer literal"
;	"End of statement"

Figure 6.14 A syntax diagram (语法图) of our if-then-else pseudocode statement

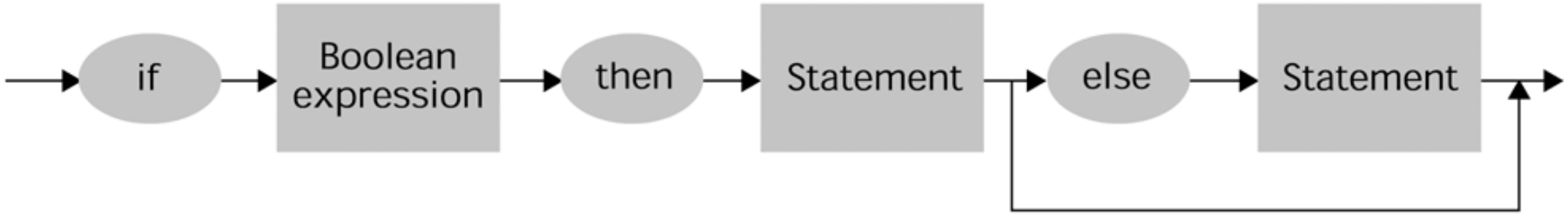
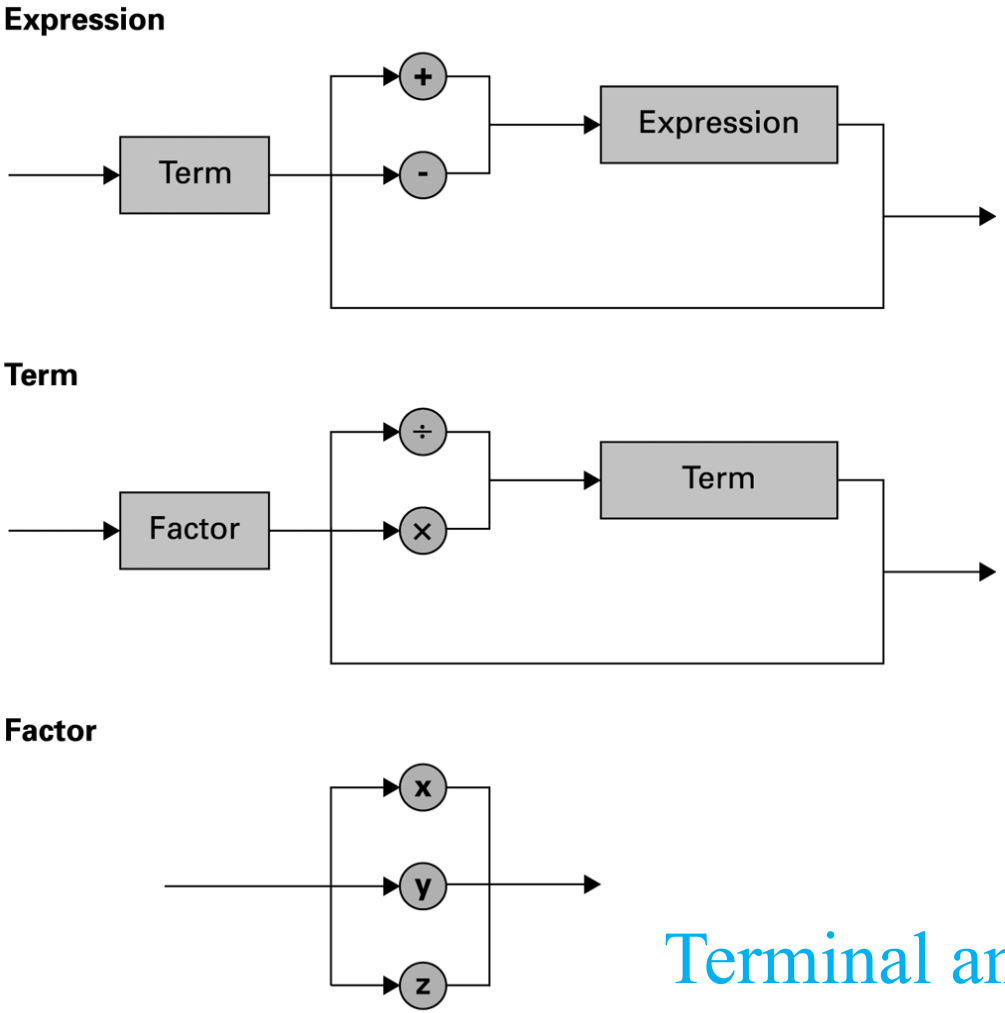
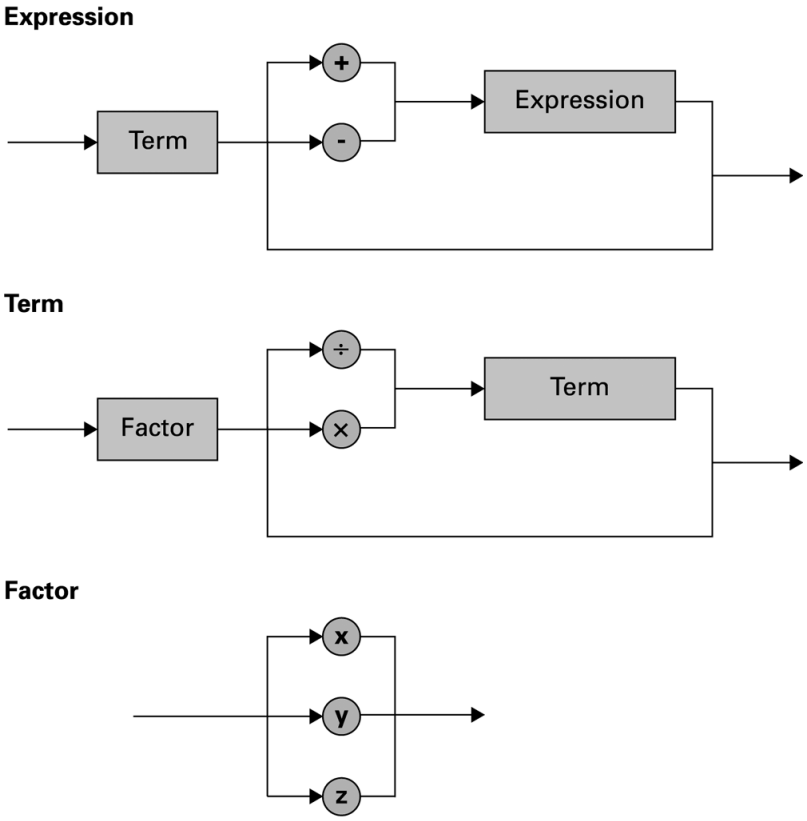


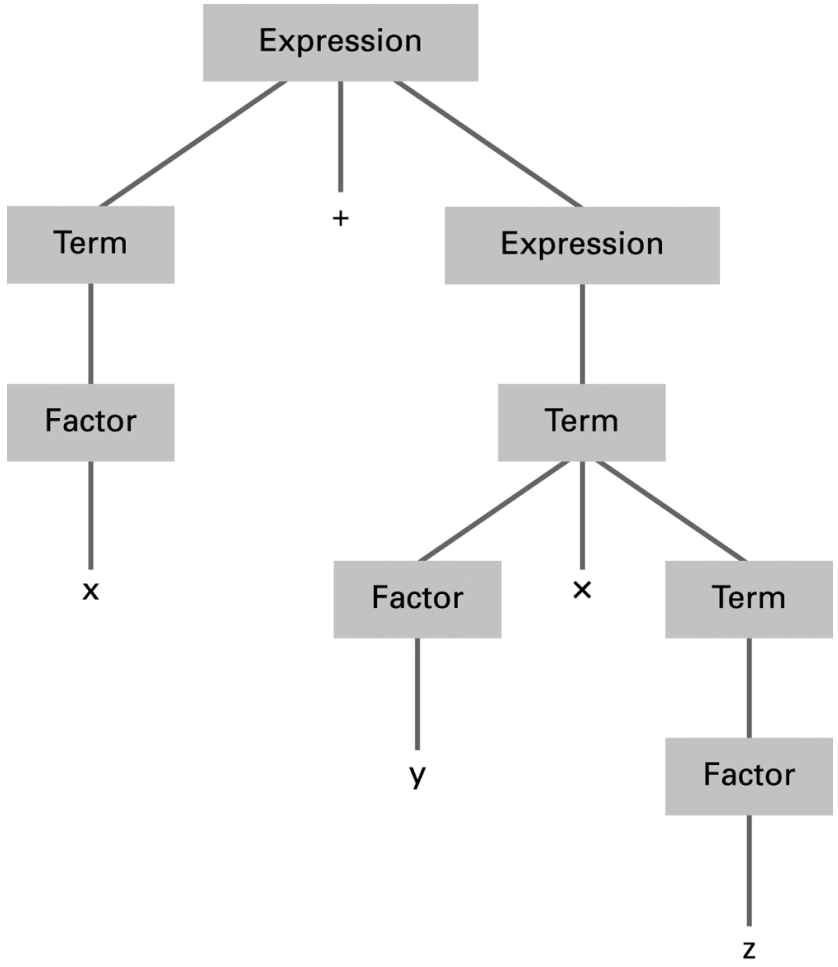
Figure 6.15 Syntax diagrams describing the structure of a simple algebraic expression



Terminal and nonterminal

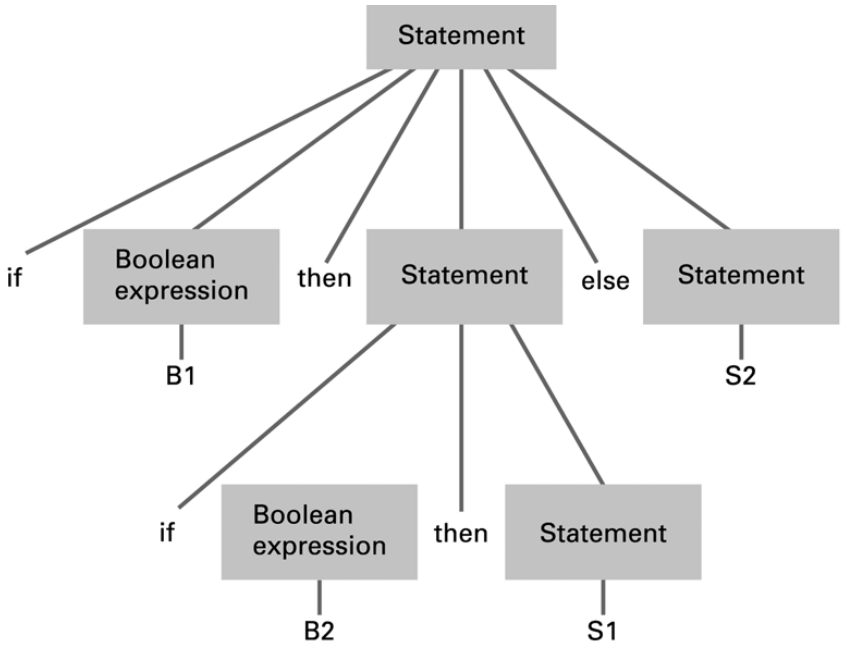
Figure 6.16 The parse tree for the string x + y x z based on the syntax diagrams in Figure 6.15

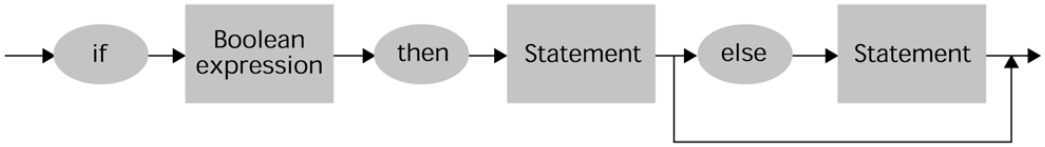


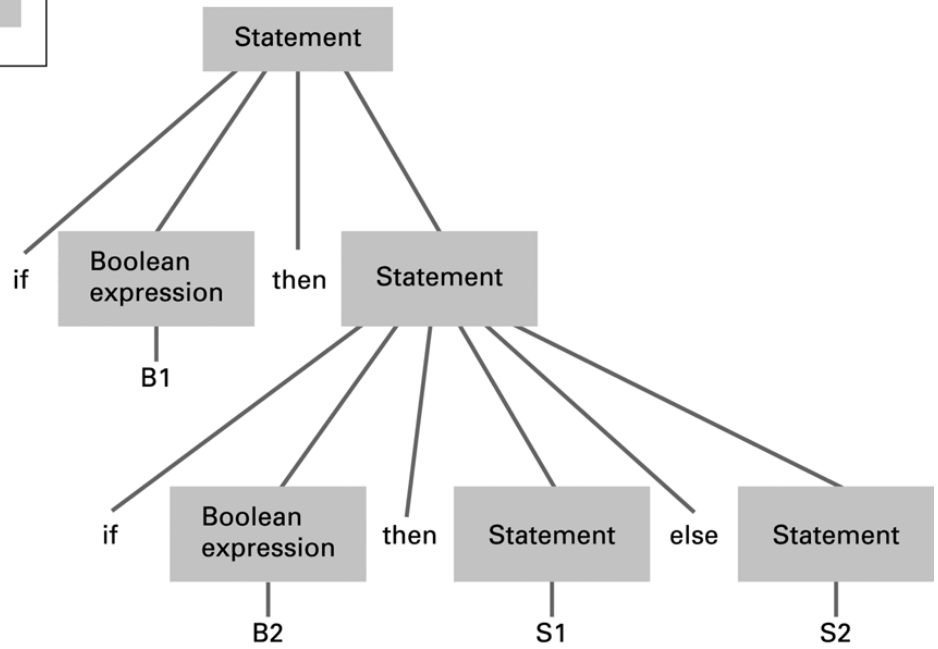


Parse trees for the statement

if B1 then if B2 then S1 else S2







- Questions?

Programming methods

- Structured programming
- Object-oriented programming

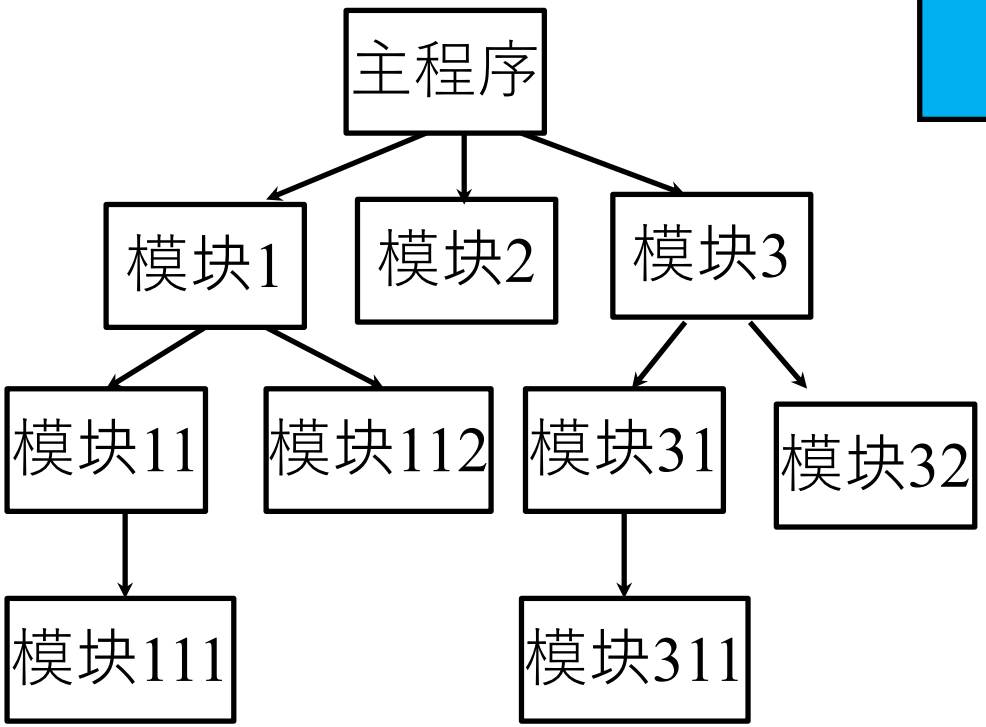
结构化程序设计思想

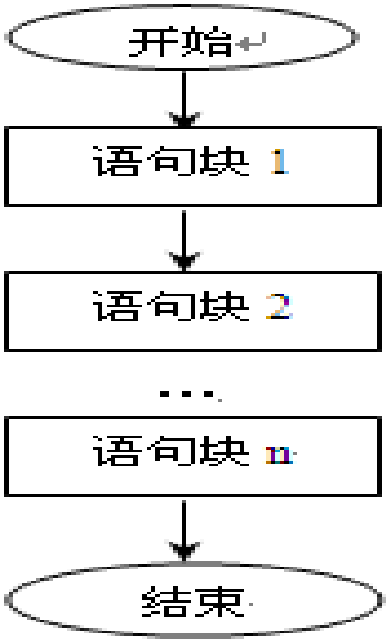
- 最早由荷兰科学家E.W.Dijkstra提出
  - 任何程序都基于顺序、选择、循环三种基本的控制结构
  - 程序具有模块化特征，每个程序模块具有惟一的入口和出口
  - 取消GOTO语句
- 结构化程序的结构简单清晰，可读性好，模块化强。

结构化编程主要包括两个方面

提倡采用自顶向下、逐步细化的模块化程序设计原则

每个模块强调采用单入口单出口的三种基本控制结构（顺序、选择、循环），避免使用GOTO语句





面向对象程序设计

- 80年代初面向对象的程序设计(Object Oriented Programming，简称OOP)
  - 用面向对象的方法解决问题，不再将问题分解为过程，而是将问题分解为对象。
- 对象：属性、方法和事件
  - “对象＋消息”的面向对象的程序设计模式有取代“数据结构＋算法”的面向过程的程序设计模式的趋向。

两者区别

- 结构化的分解突出过程：
  - 如何做(How to do)？它强调代码的功能是如何得以完成。
- 面向对象的分解突出真实世界和抽象的对象：
  - 做什么(What to do)？它将大量的工作由相应的对象来完成，程序员在应用程序中只需说明要求对象完成的任务。

面向对象程序设计益处

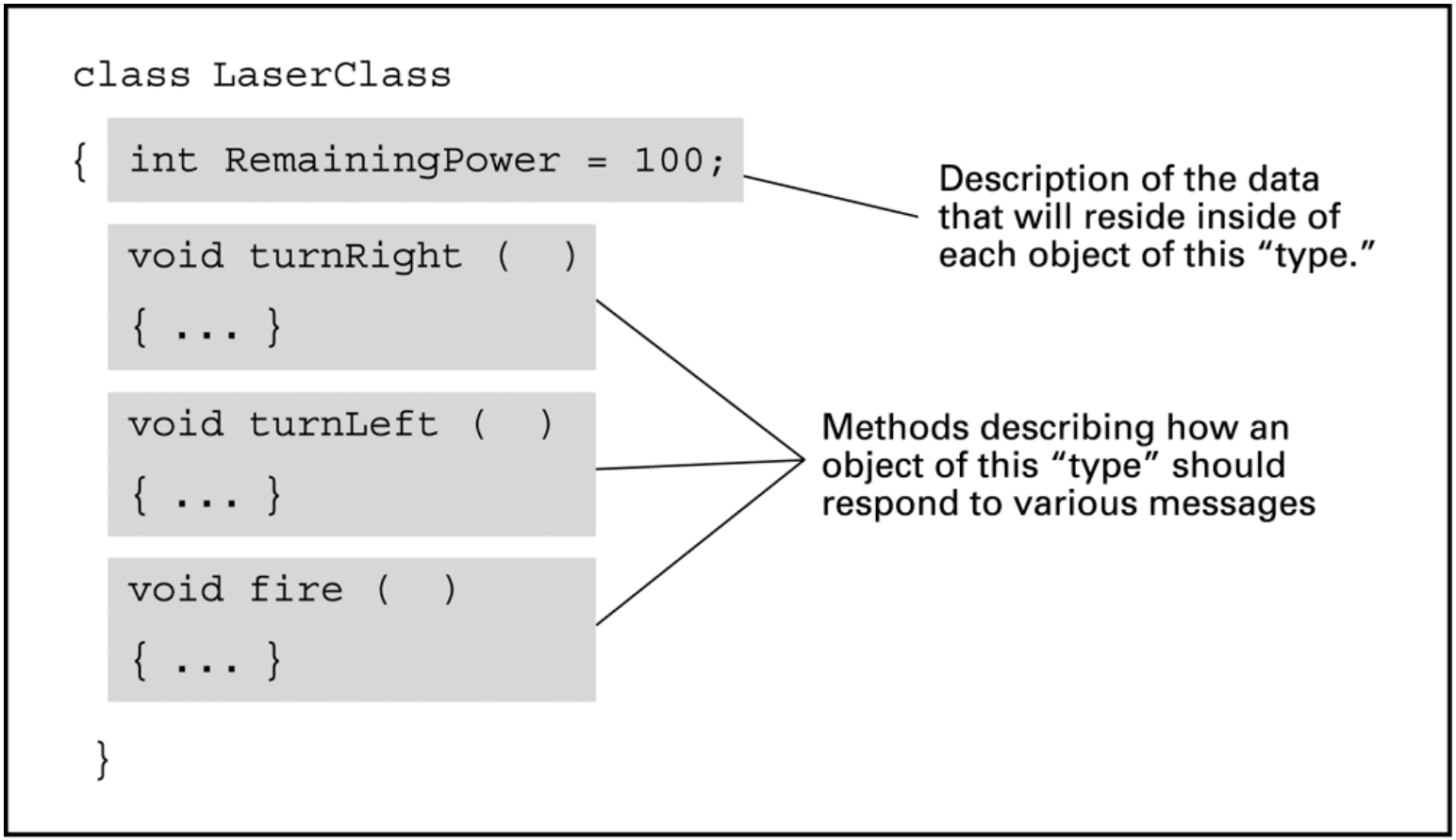
① 符合人们习惯的思维方法，便于分析复杂而多变化的问题；

② 易于软件的维护和功能的增减；

③ 可重用性好，能用继承的方式减短程序开发所花的时间；

④ 与可视化技术相结合，改善了工作界面和便于与用户交互。

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



Objects and Classes

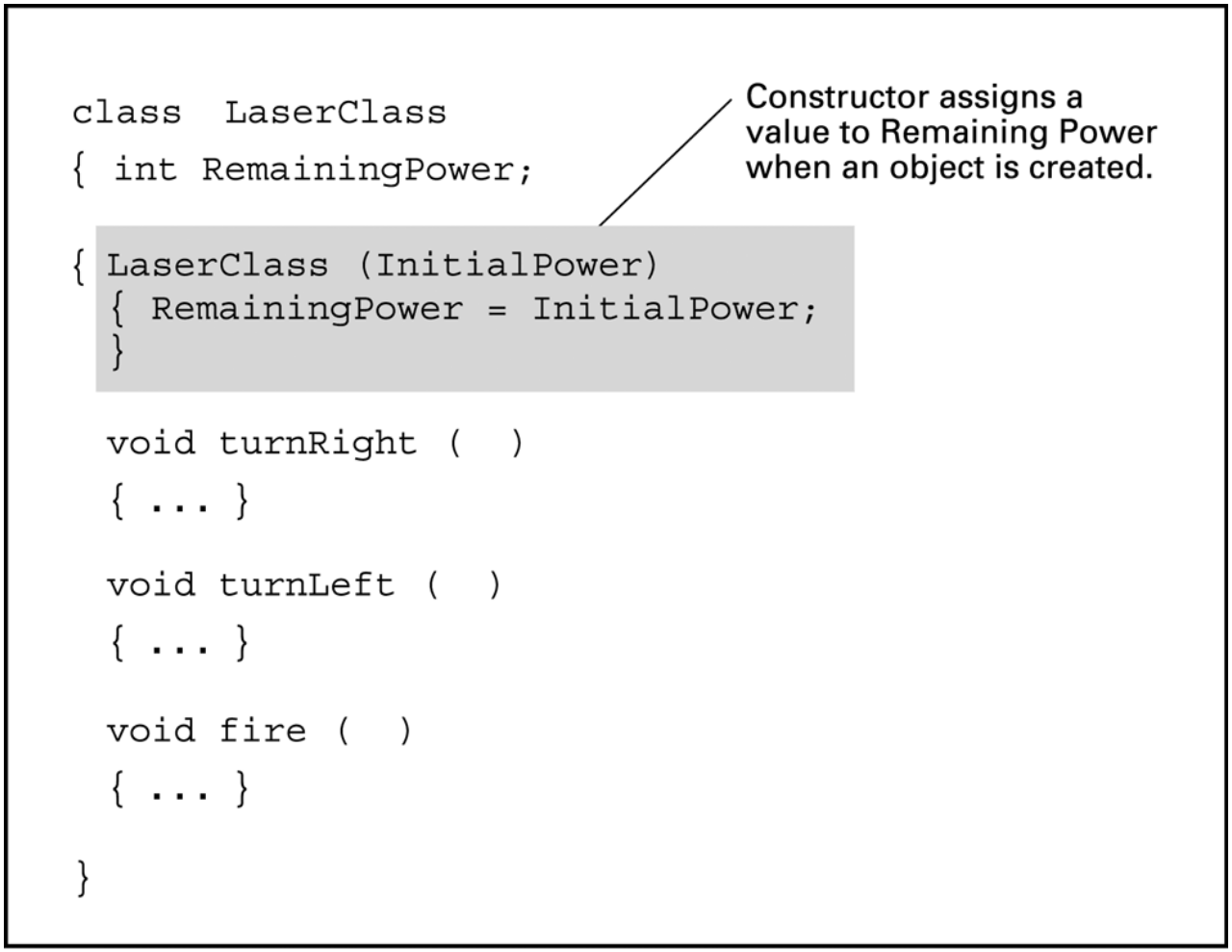
- **Object:** Active program unit containing both data and procedures
- **Class:** A template from which objects are constructed

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

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

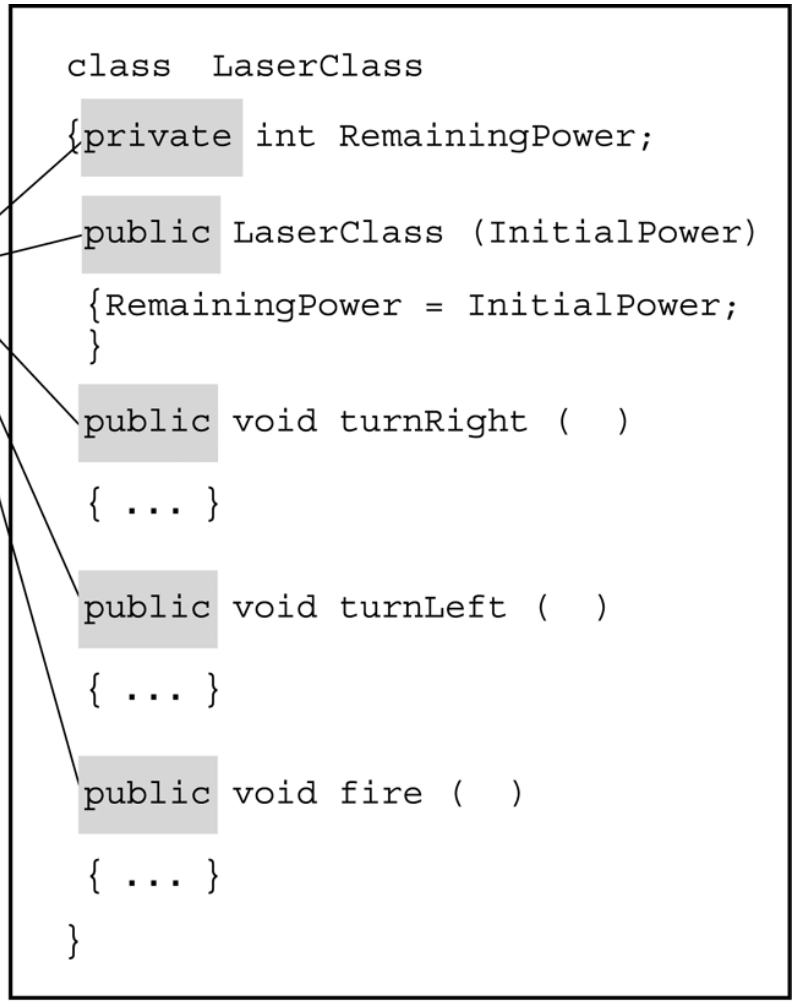


Object Integrity

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

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

Components in the class are designated public or private depending on whether they should be accessible from other program units.



```
class  LaserClass
{private int RemainingPower;

public LaserClass (InitialPower)
 {RemainingPower = InitialPower;
 }

public void turnRight (  )

 { ... }

public void turnLeft (  )

 { ... }

public void fire (  )

 { ... }
}
```

Object-oriented programming

- Class
  - Property
  - Methods
- Instance
- Object

```
Class Shape
{
    private:
        string My_name;

    public:
        Shape(){}
        void set_name (string name)
            { My_name = name; }
        void Hello()
            {cout<<“I am a shape”;}
} ;
```

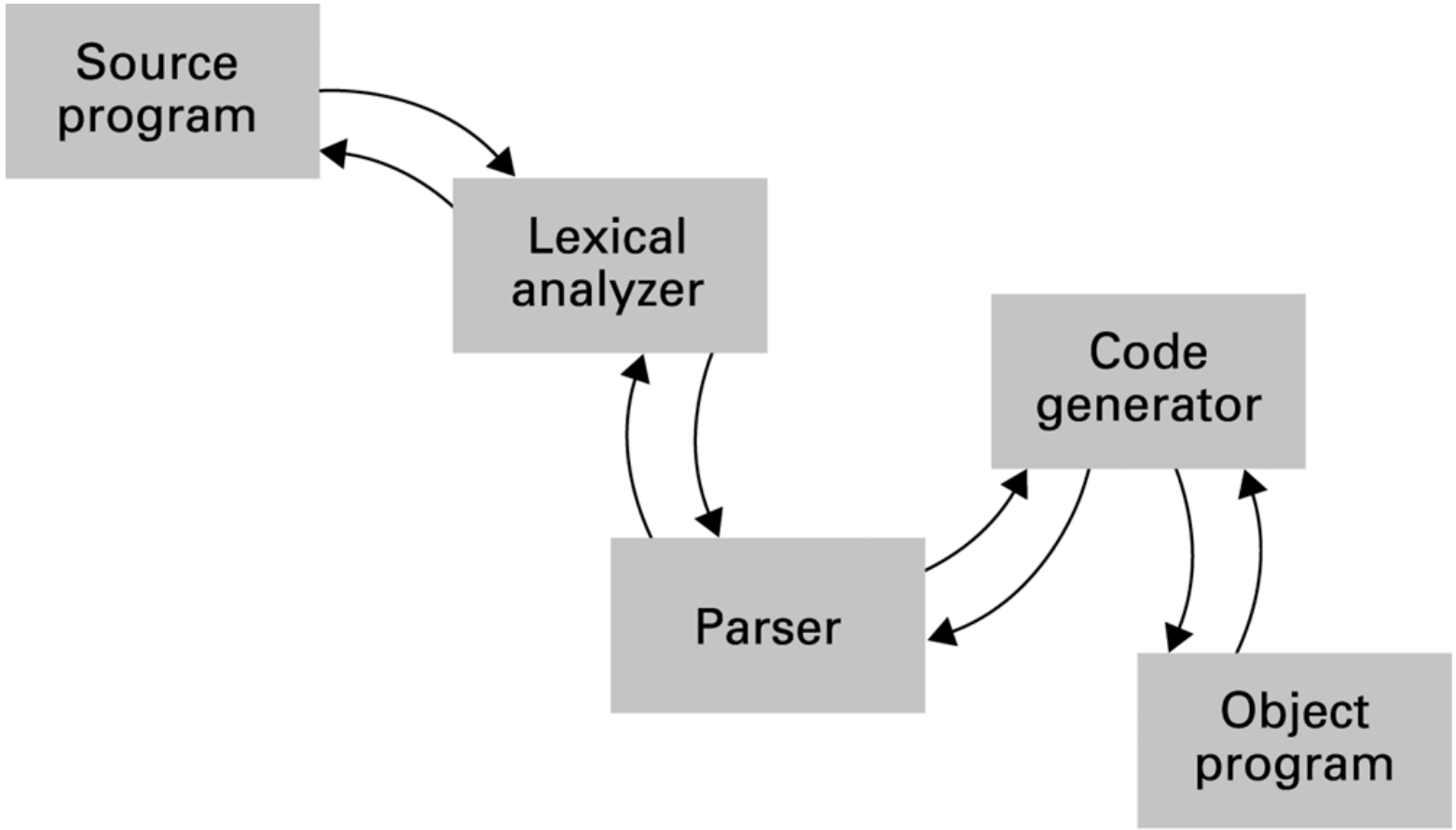
Which one is incorrect?

- Shape myShape;
- Shape.My_name = “Shape A”;
- Shape.Hello();
- myShape.My_name = “Shape A”;
- myShape.set_name(“Shape A”);
- myShape.Hello();

```
Class Shape
{
   private:
      string My_name;

   public:
        Shape(){}
        void set_name (string name)
            { My_name = name; }
        void Hello()
              {cout<<“I am a shape”;}
} ;
```

Figure 6.18 An object-oriented approach to the translation process



Additional Object-oriented Concepts

- Encapsulation 封装
- **Inheritance 继承**
- **Polymorphism多态**

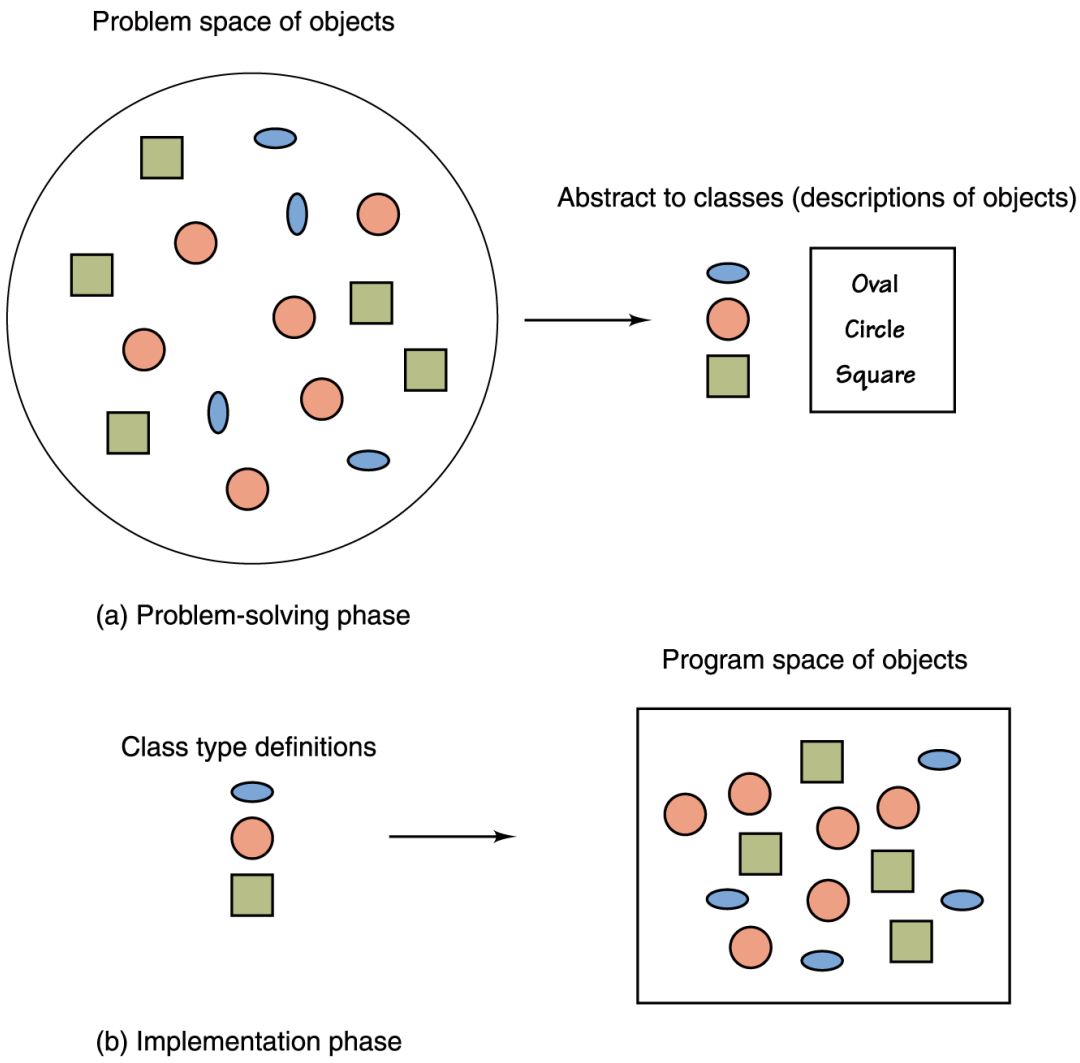
Inheritance and polymorphism

- Inheritance

```
Class Line : public Shape
{
   public:
      Line(){}
}
```

- Which one is incorrect?
  - Line myLine;
  - myLine.My_name = “Line B”;
  - myLine.set_name (“Line B”);
  - myLine.Hello();

Inheritance



- Inheritance and polymorphism combined allow the programmer to build useful hierarchies of classes that can be reused in different applications

Figure
Mapping of problem into solution

Inheritance and polymorphism

- Line myLine;
- myLine.Hello()
- Square mySquare;
- mySquare. Hello()
- Shape* myShape = &myLine;
- myShape->Hello()

```
Class Line : public Shape
{
   public:
      Line(){}
      void Hello()
         {cout<<"I am a Line";}
}
Class Square: public Shape
{
    public:
       Square(){}
       void Hello()
          {cout<<"I am a Square";}}
}
```

Polymorphism

- Shape* myShape = &myLine;
- myShape->Hello()

```
Class Shape
{
    private:
        string My_name;

    public:
        Shape(){}
        void set_name (string name)
            { My_name = name; }
        virtual void Hello()
            {cout<<"I am a shape";}
}
```

Polymorphism

- Shape* = new Shape[2];
- Shape[0] = &myLine;
- Shape[1] = &mySquare;
- for(int i = 0; i < 2; i++)
- {
  Shape[i] -> Hello();
  Shape[i] -> Draw();
- }

- Questions?

Universal Programming Language

A language with which a solution to any computable function can be expressed

- Examples: “Bare Bones” and most popular programming languages

The Bare Bones Language

- Bare Bones is a simple, yet universal language.
- Statements
  - **`clear` *`name`*`;`**
  - **`incr` *`name`*`;`**
  - **`decr` *`name`*`;`**
  - **`while` *`name`* `not 0 do; ... end;`**

Figure 12.4 A Bare Bones program for computing X x Y

```
clear Z;
while X not 0 do;
   clear W;
   while Y not 0 do;
      incr Z;
      incr W;
      decr Y;
   end;
   while W not 0 do;
      incr Y;
      decr W;
   end;
   decr X;
end;
```

Figure 12.5 A Bare Bones implementation of the instruction "copy Today to Tomorrow"

```
clear Aux;
clear Tomorrow;
while Today not 0 do;
    incr Aux;
    decr Today;
end;
while Aux not 0 do;
    incr Today;
    incr Tomorrow;
    decr Aux;
end;
```

The Most Important Open Problem In Programming Languages*

Increasing Programmer Productivity

- Write programs correctly
- Write programs quickly
- Write programs easily

- Why?
  - Decreases support cost
  - Decreases development cost
  - Decreases time to market
  - Increases satisfaction

Influences on programming languages

- Computer capabilities
  - Hardware and OS
- Applications
  - Wide area of applications
- Programming methods
  - Multiprogramming, interactive systems, data abstraction, formal semantics,O-O programming,…
- Implementation methods
- Theoretical studies
- Standardization

- Questions?

Key points

- Programming paradigms
- Variables and data types
- Data structure, constants and literals
- Assignment, control and comments
- Procedure, parameters, function
- Translation process
- Object-oriented programming: class, objects, constructor, additional features