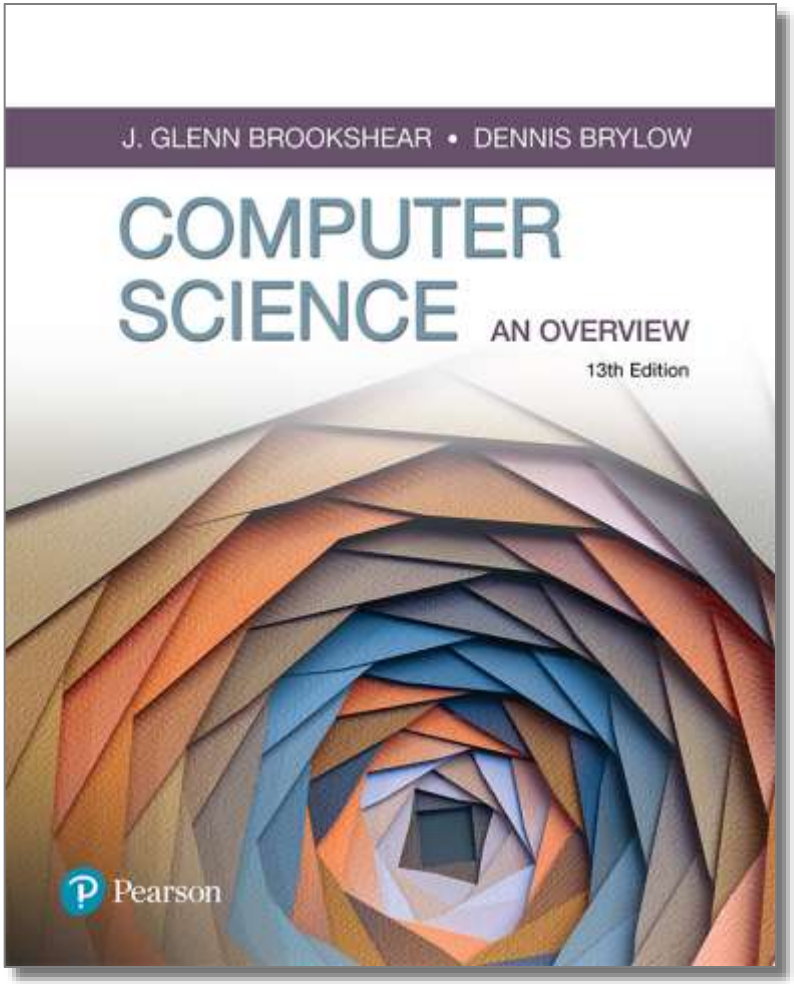
# Computer Science An Overview

13th Edition




## Chapter 6

Programming Languages

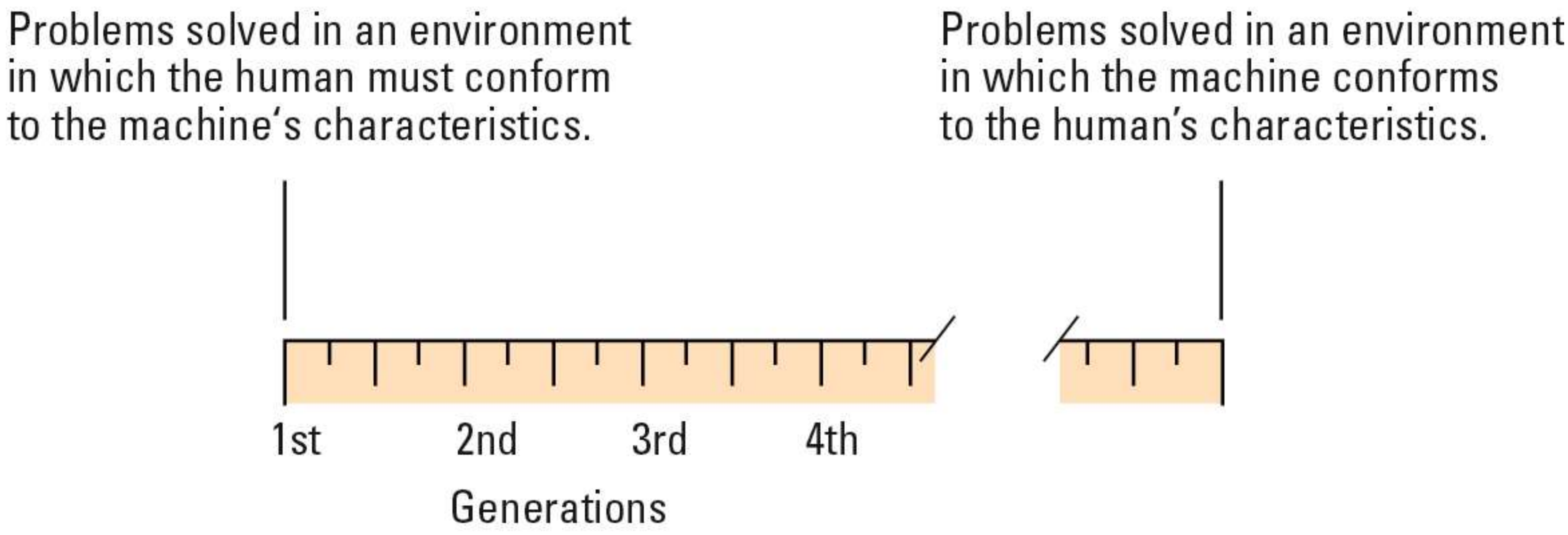
# Chapter 6: Programming Languages

- 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 | Assembly language |
| --- | --- |
| 156C | LD R5, Price |
| 166D | LD R6, ShipCharge |
| 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
```

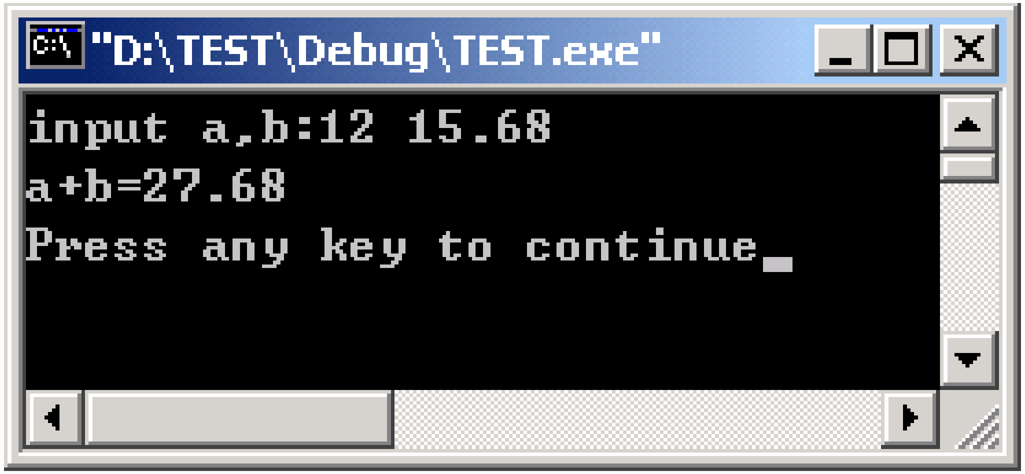
**Object code**

# Third Generation Language

- Uses high-level primitives
  - Similar to our pseudocode in Chapter 5
- Machine independent (mostly)
- Examples: C++
- 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

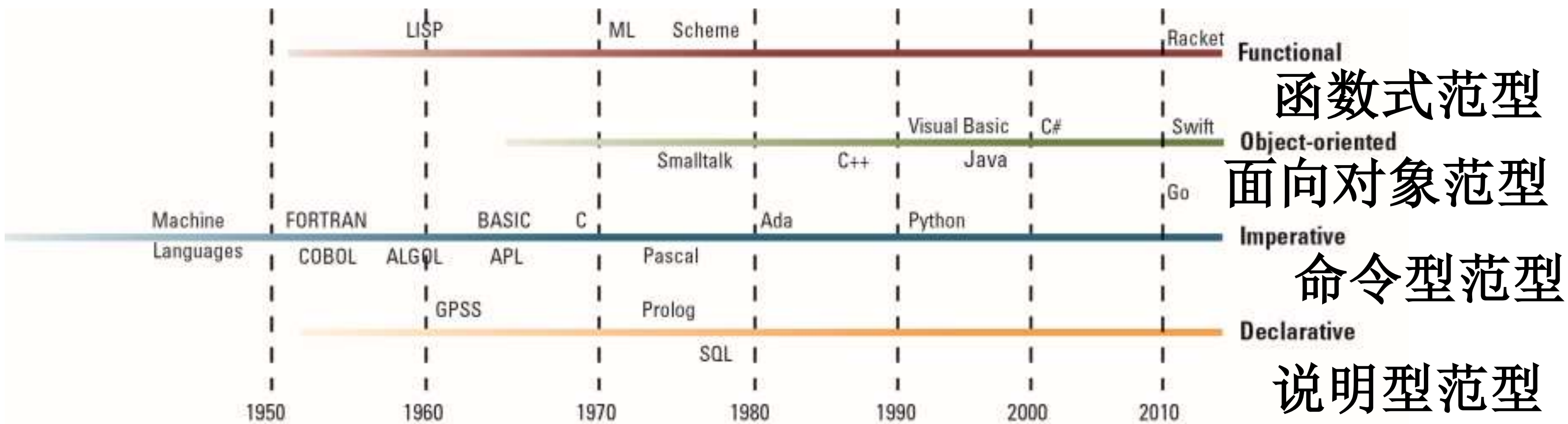
- SQL

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)

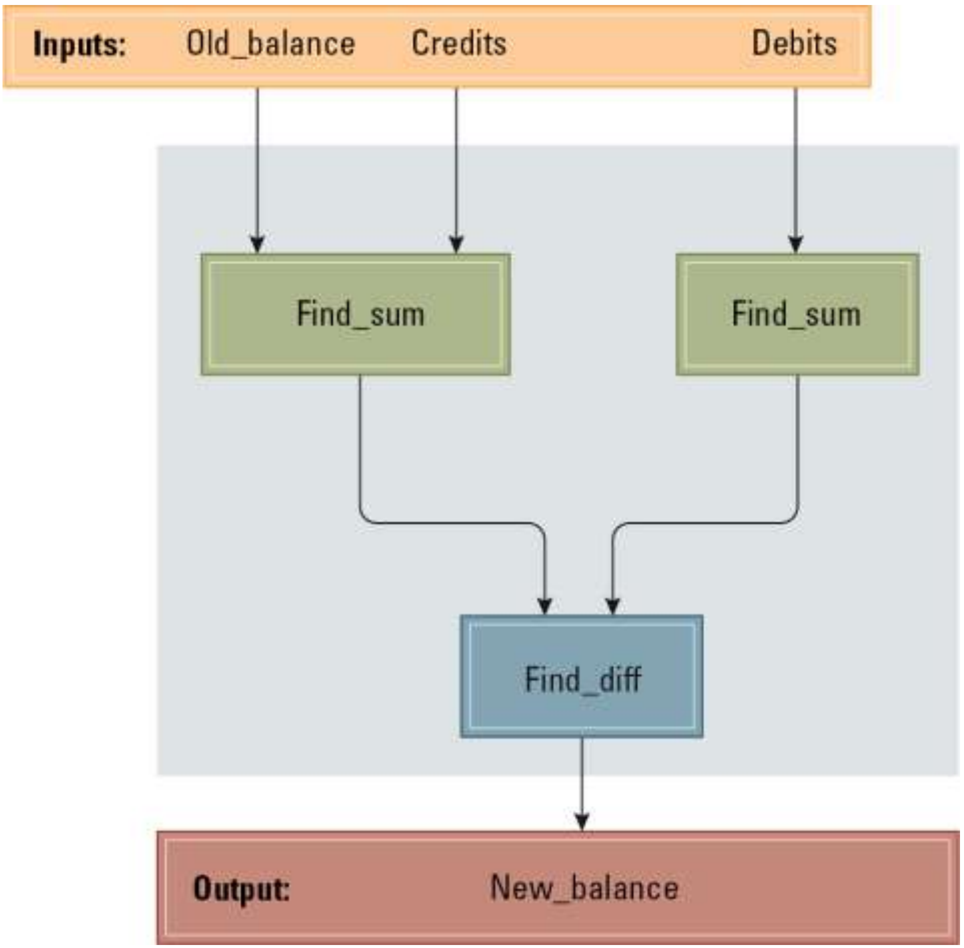
update students set score=55

select student_name, major, scholarship from students

# Figure 6.2 The evolution of programming paradigms



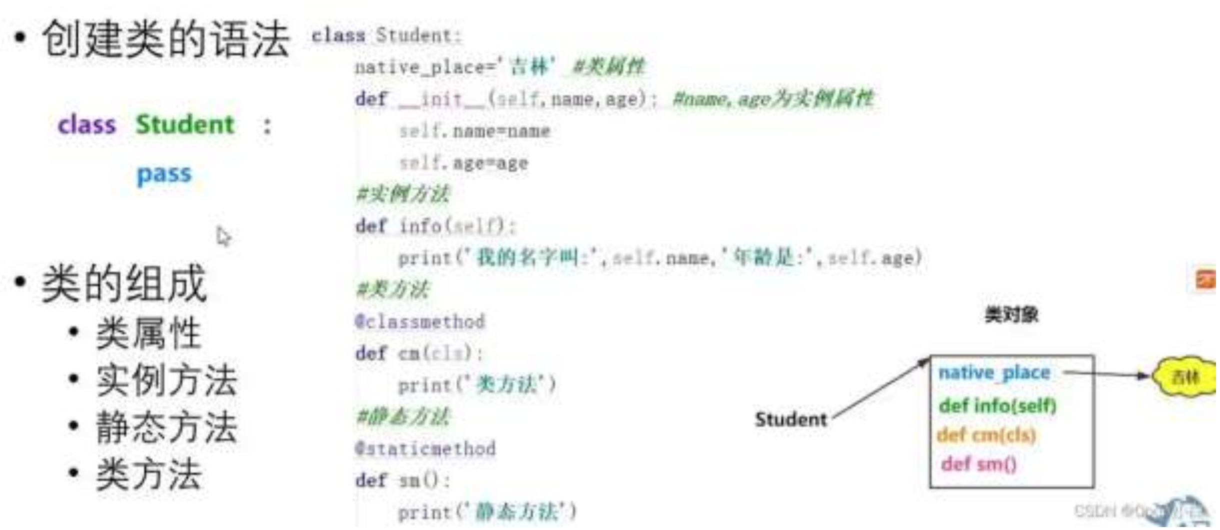
# Figure 6.3 A function for checkbook balancing constructed from simpler functions



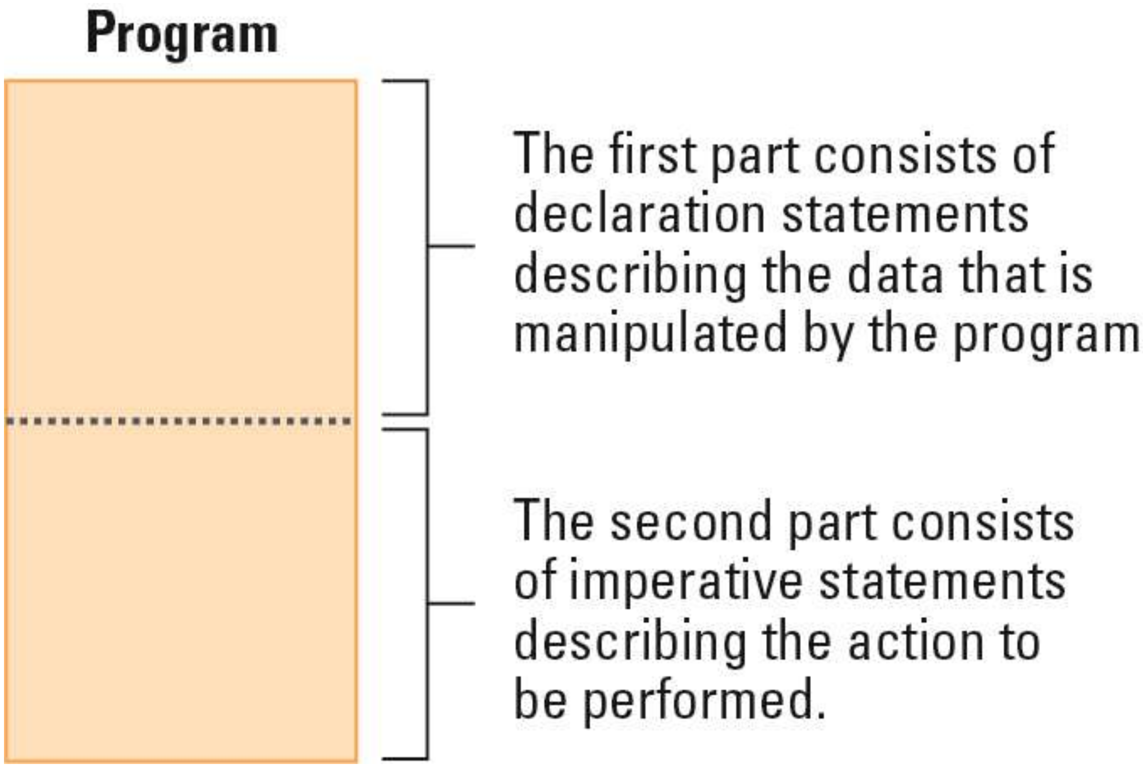

函数式范型将计算过程表达为一系列函数调用，通过函数来处理数据。程序可以看作是接受输入和产生输出的实体。

# 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;
}
```

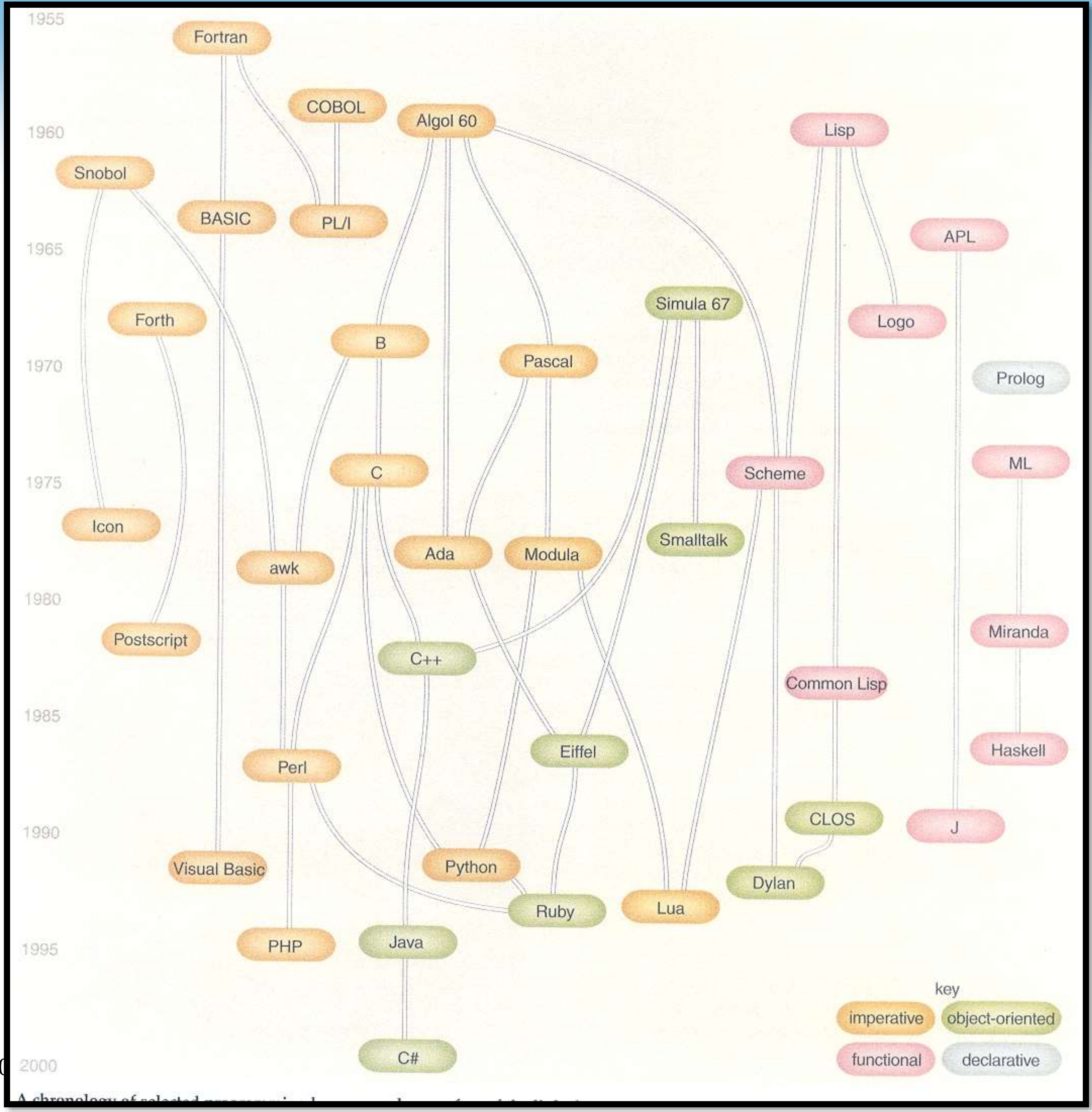
命令型范型
(面向过程) 开发一个命令序列，遵照这个序列，对数据进行操作以产生所期望的结果

# 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

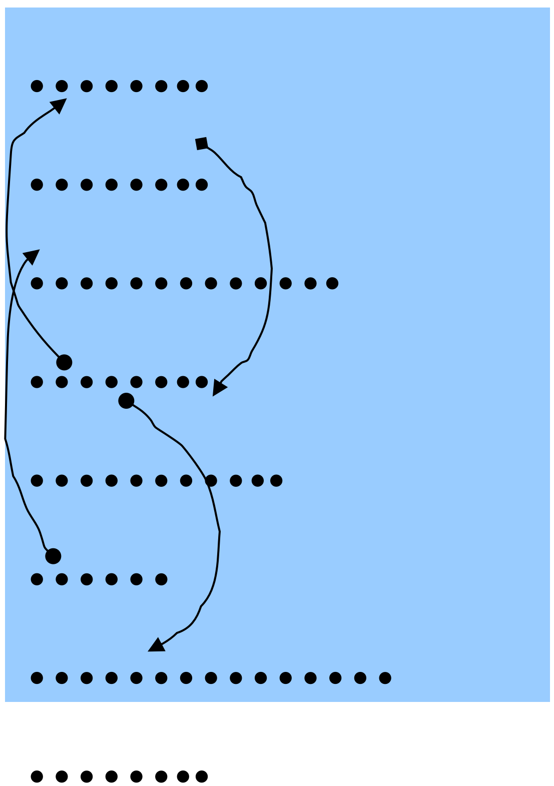

1955
1960
1965
1970
1975
1980
1985
1990
1995
2000
Fortran
COBOL
Algol 60
Lisp
Snobol
BASIC
PL/I
APL
Forth
Simula 67
Logo
B
Pascal
Prolog
ML
C
Scheme
Icon
Smalltalk
Ada
Modula
awk
Postscript
Miranda
C++
Common Lisp
Haskell
Eiffel
Perl
CLOS
J
Python
Visual Basic
Dylan
Ruby
Lua
PHP
Java
C#
key
imperative
object-oriented
functional
declarative

# 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

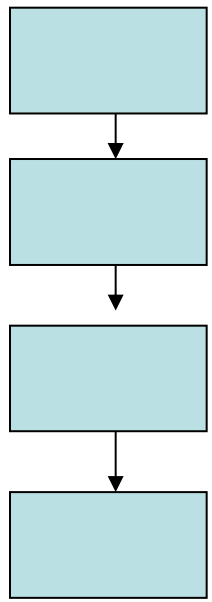
世博会远大馆 2000平米、6层楼的建筑，工人用24小时建成--

## 初期的程序设计



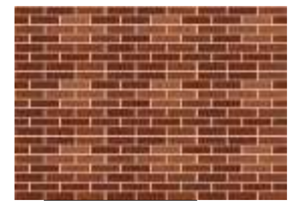
一碗面条式程序（BS）

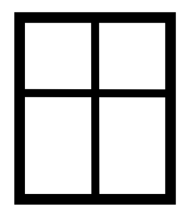
## 结构化程序设计



一串珠子式串连成

## 面向对象程序设计





拼装

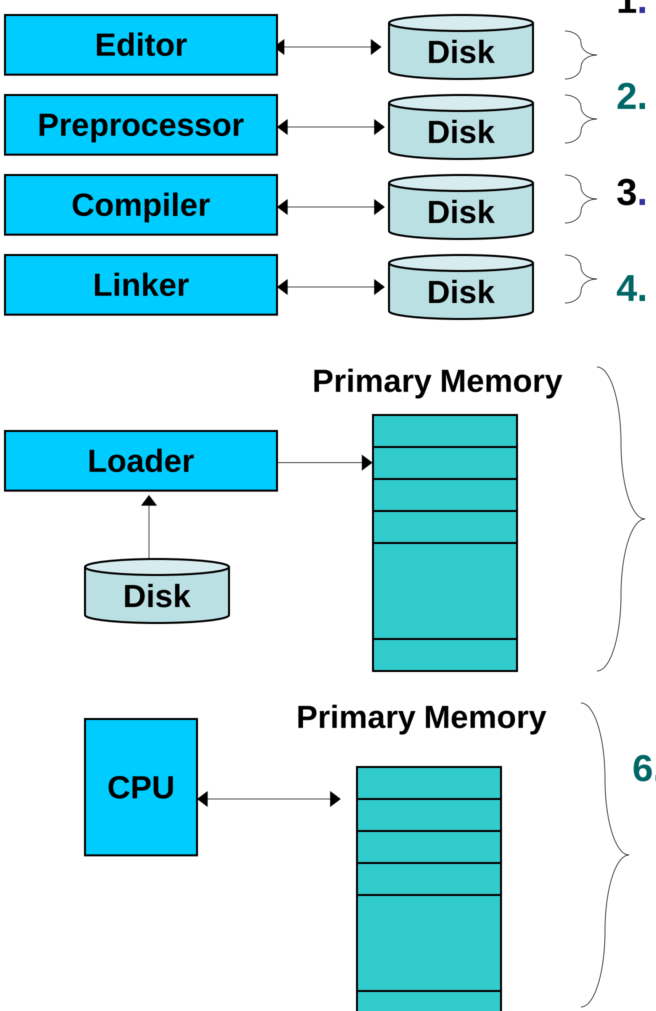
搭积木式

# 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

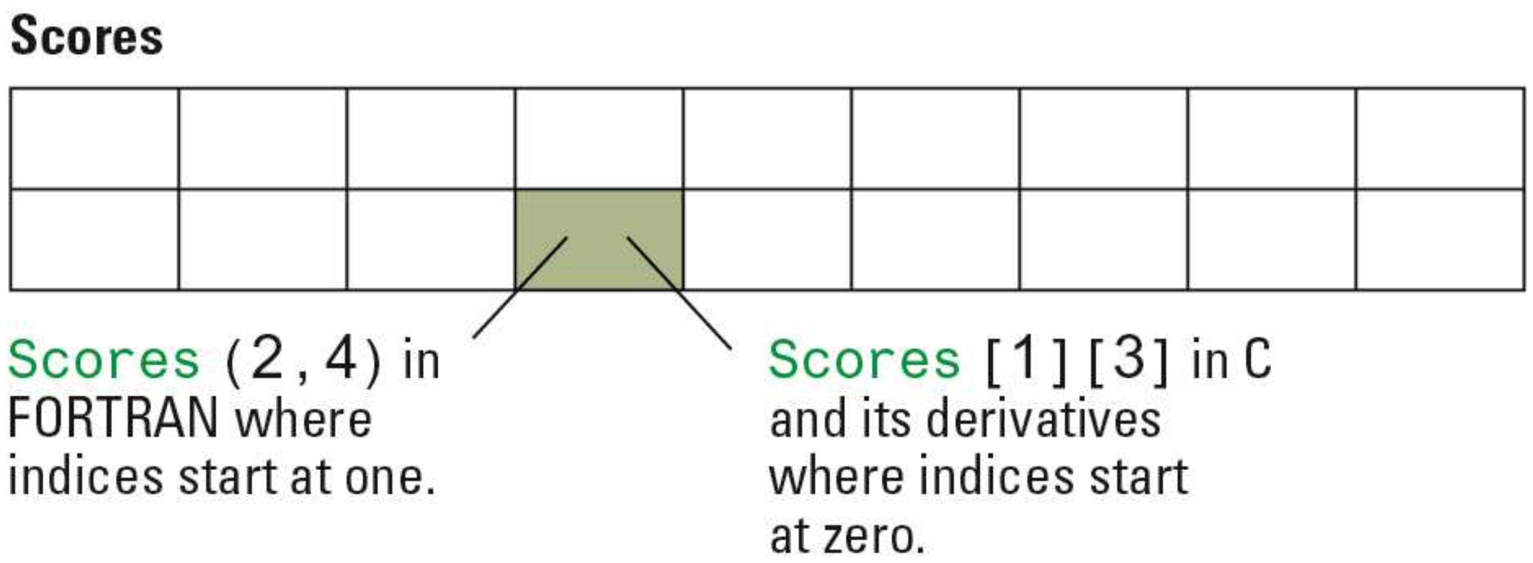
- Conceptual shape or arrangement of data
- A common data structure is the **array**
  - In C

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

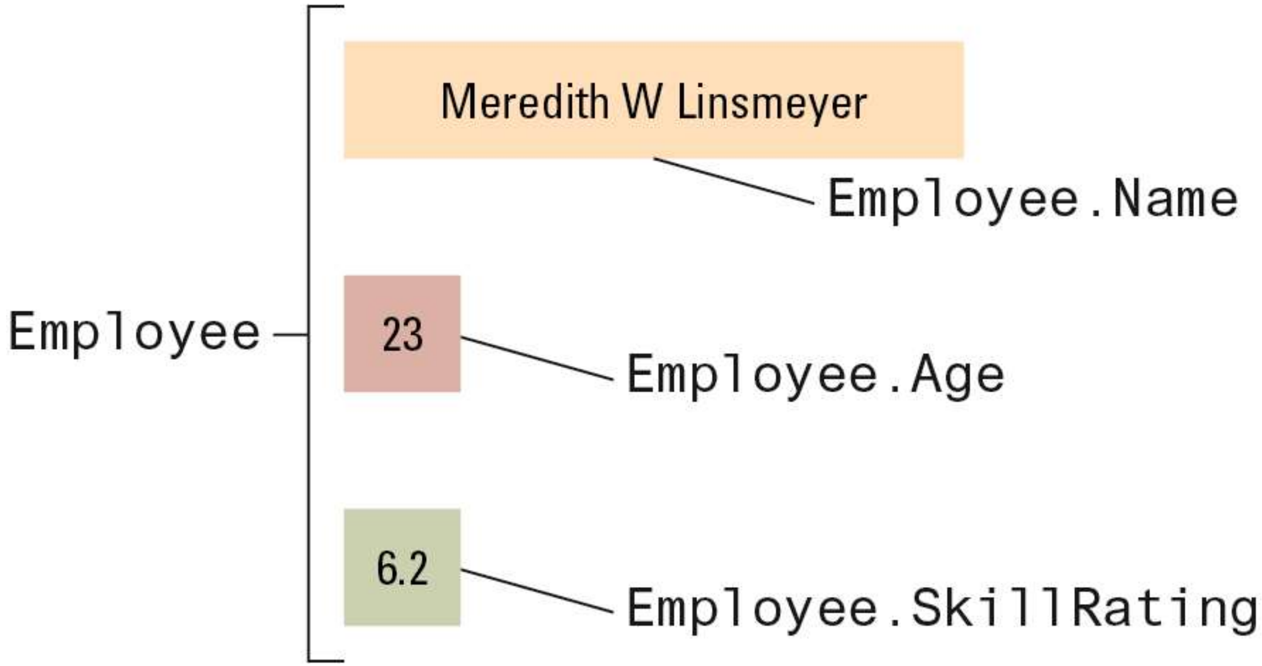
  - In FORTRAN

```
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)

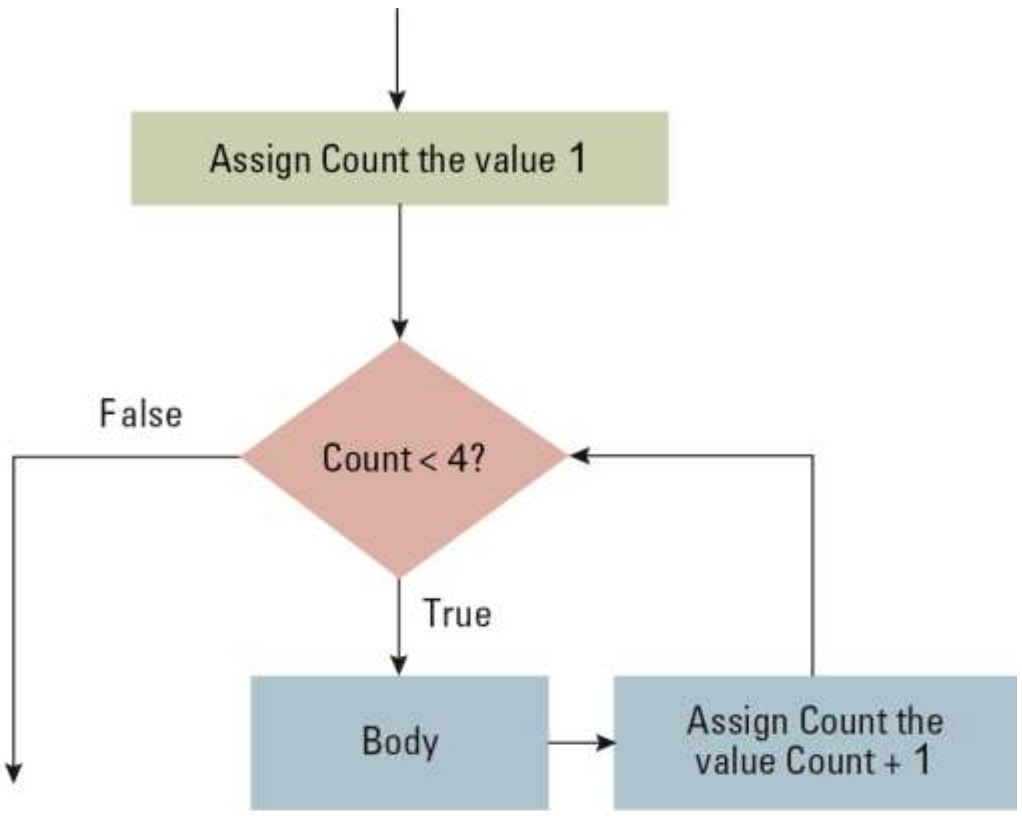
- Switch statement in C, C++, C#, and Java

```
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

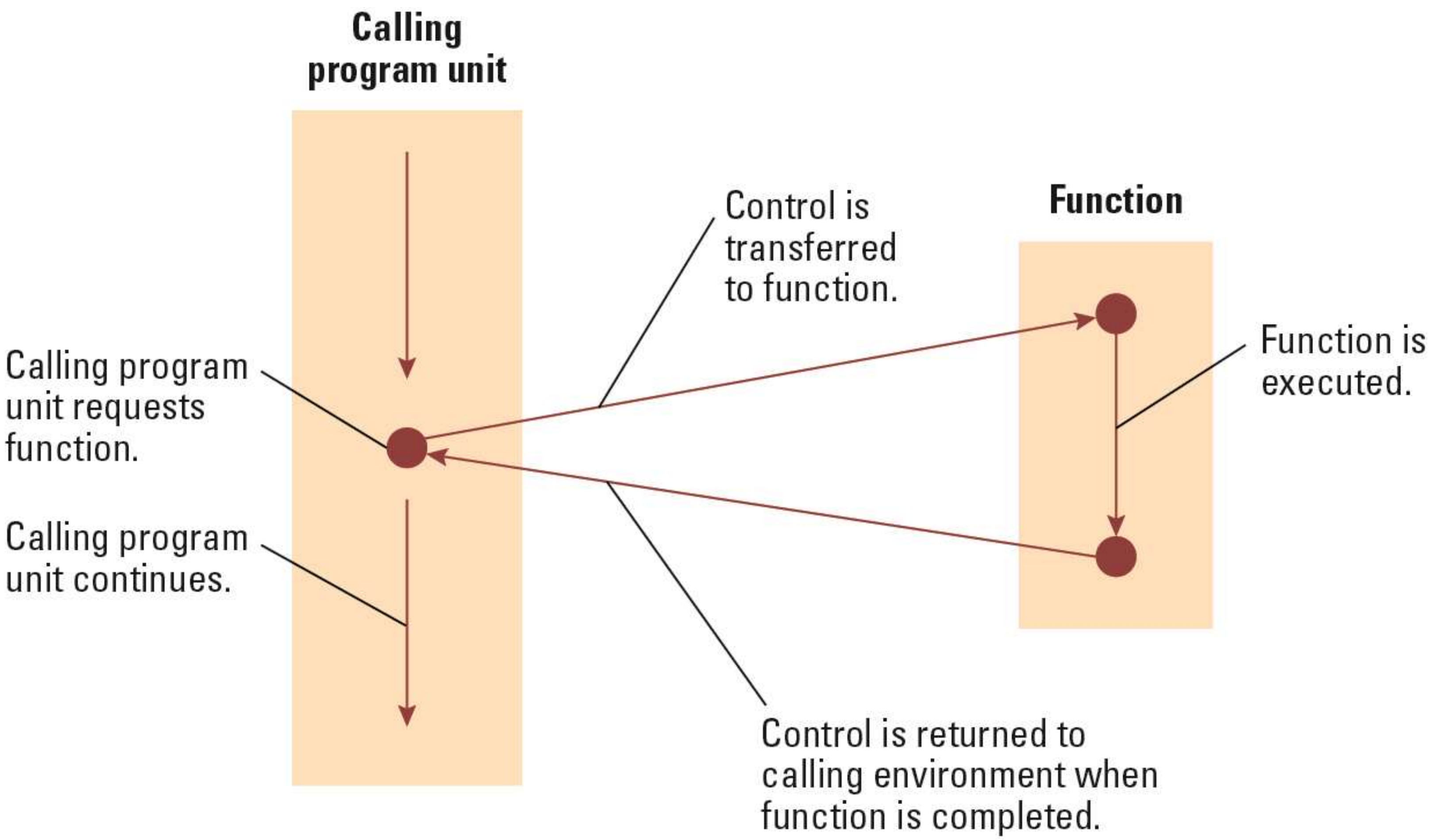
- Explanatory statements within a program
- Helpful when a human reads a program
- Ignored by the compiler

```
/* 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

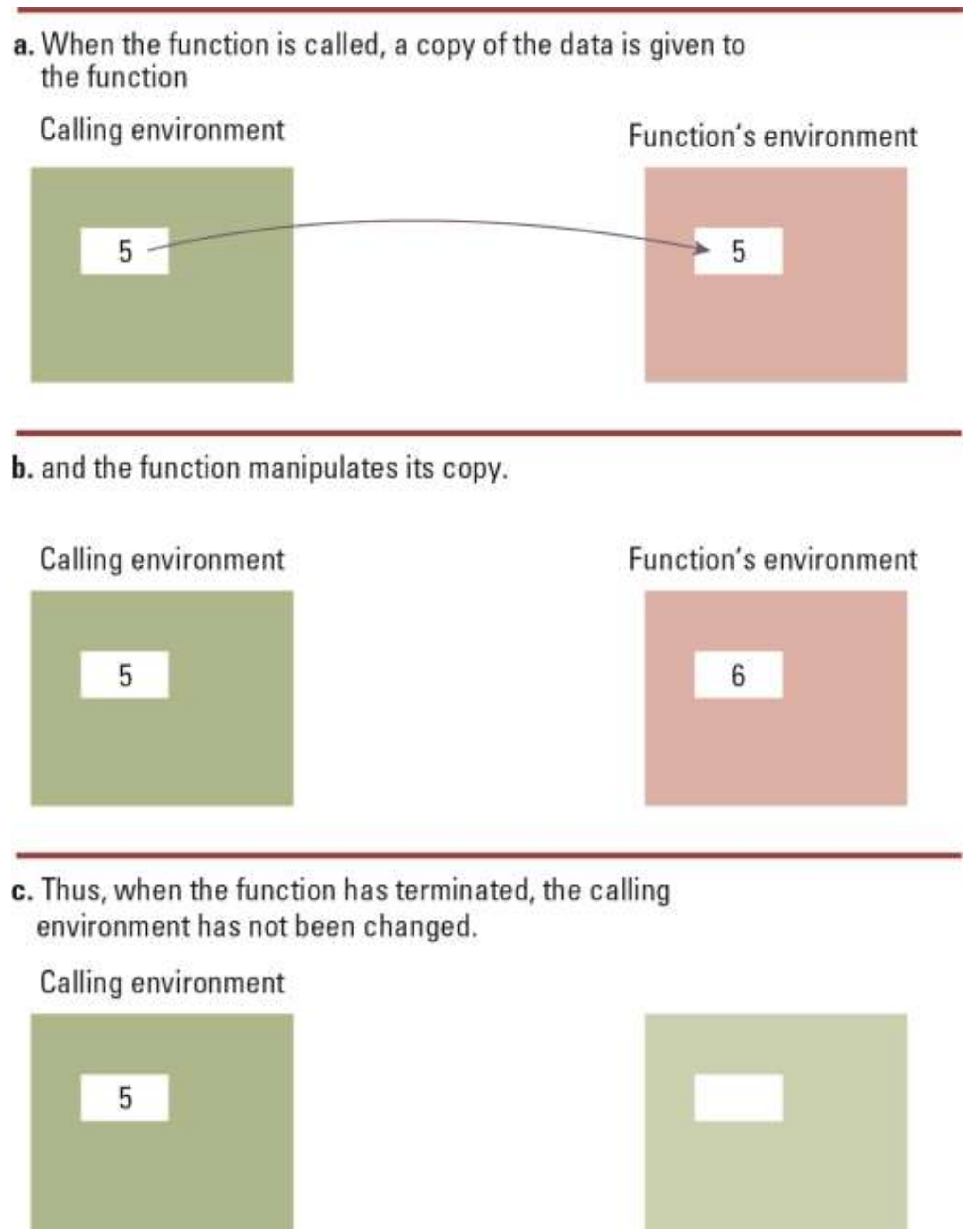
Starting the header with the term "void" is the way that a C programmer specifies that the program unit returns no value. We will learn about return values shortly.

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

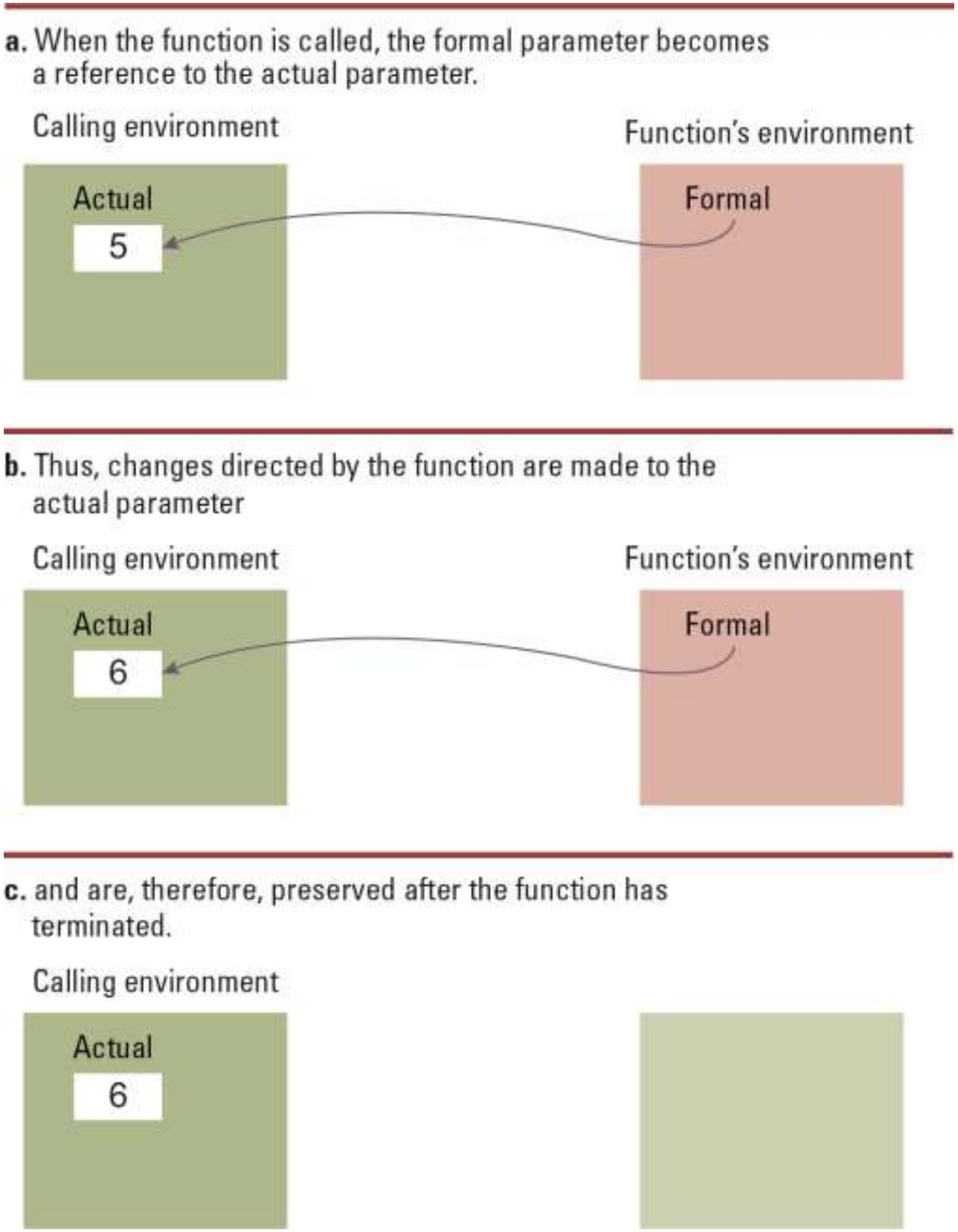
```
void ProjectPopulation (float GrowthRate)

{     int Year;              // This declares a local variable named Year.

      Population[0] = 100.0;
      for (Year = 0; Year =< 10; Year++)
      Population[Year+1] = Population[Year] + (Population[Year] * GrowthRate);
}
```

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

# 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

The function header begins with the type of the data that will be returned.

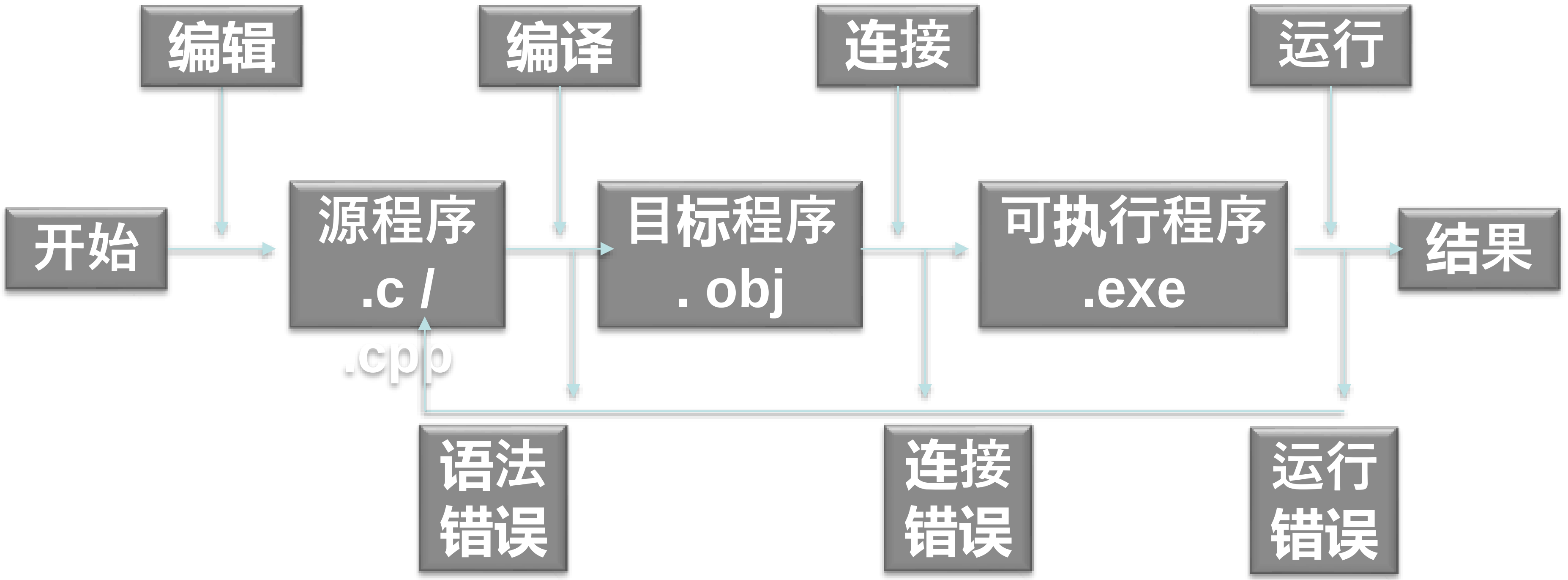
```
float CylinderVolume (float Radius, float Height)

{float Volume;

Volume = 3.14 * Radius * Radius * Height;

return Volume;

}
```

- `{float Volume;` — Declare a local variable named Volume.
- `Volume = 3.14 * Radius * Radius * Height;` — Compute the volume of the cylinder.
- `return Volume;` — Terminate the function and return the value of the variable Volume.

Cost=CostPerVolUnit*CylinderVolume(3.35,12.7)

# 6.4 Language Implementation

# 程序的调试、运行步骤

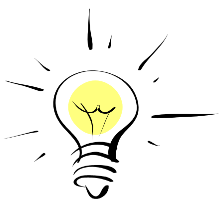


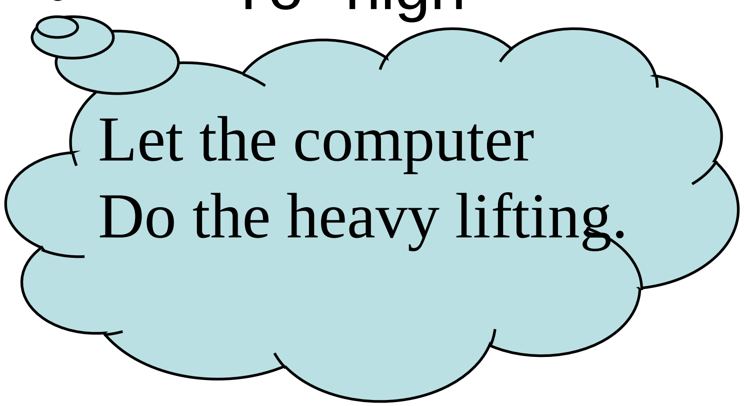
# 编译与解释

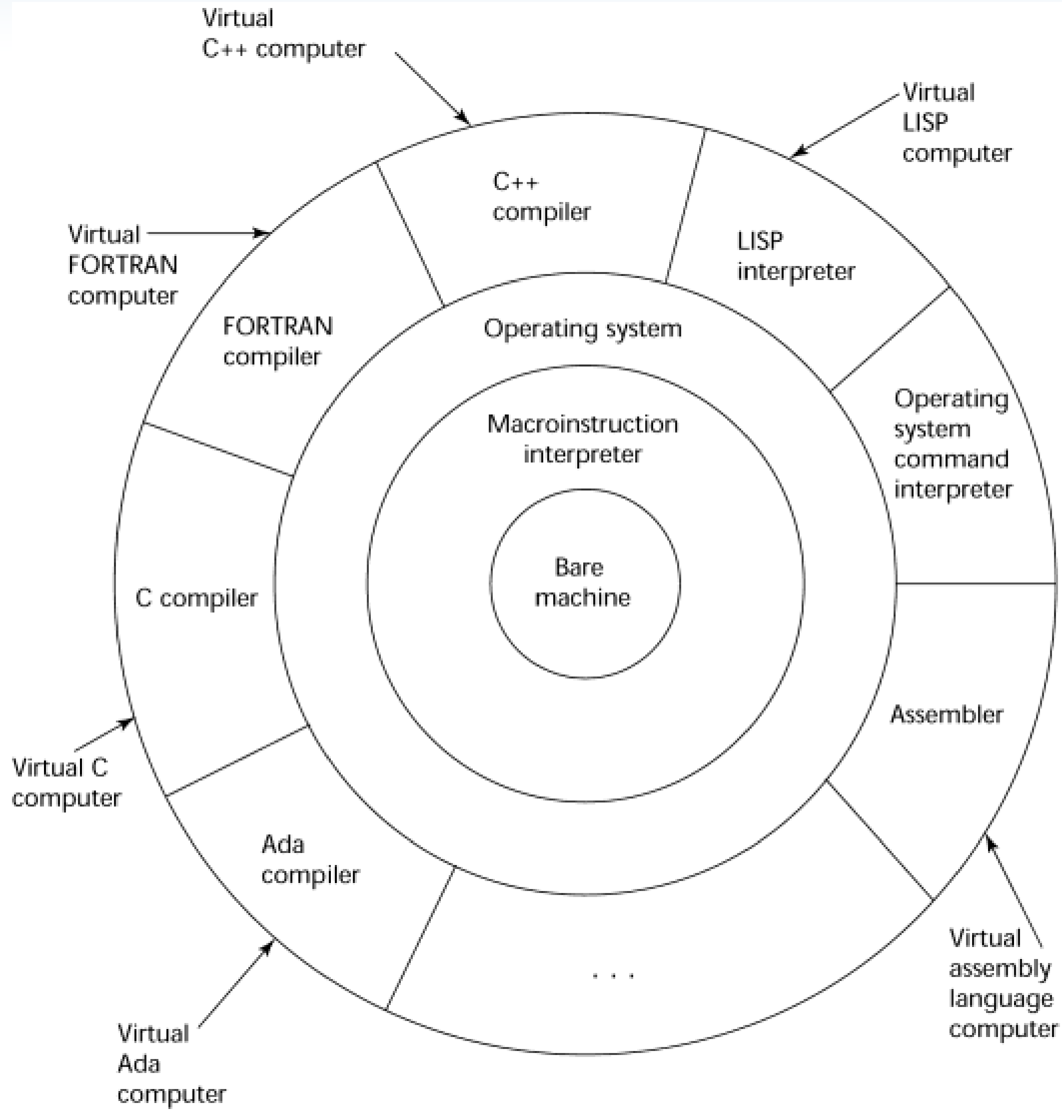
- 编译器是把源程序的每一条**语**句都**编译**成机器**语**言,并保存成二**进**制文件,**这样**运行**时计**算机可以直接以机器**语**言来运行此程序,速度很快 （如：C程序）
- 解释器**则**是只在**执**行程序**时**,才一条一条的解**释**成机器**语**言**给计**算机来**执**行,所以运行速度是不如**编译**后的程序运行的快的 （如：Python程序）

# Translate… Why?

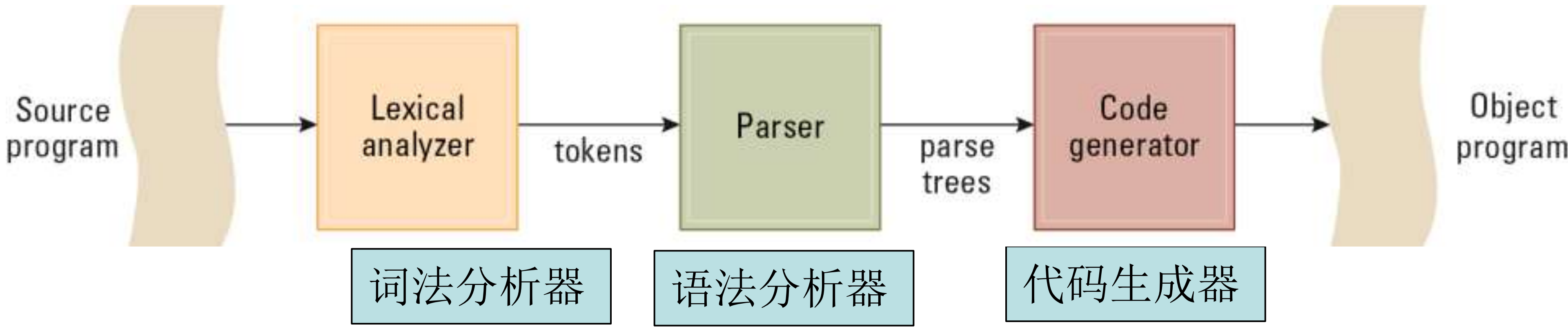
- Languages offer
  - Abstractions
  - At different levels
    - From low
      - Good for machines….
    - To  high





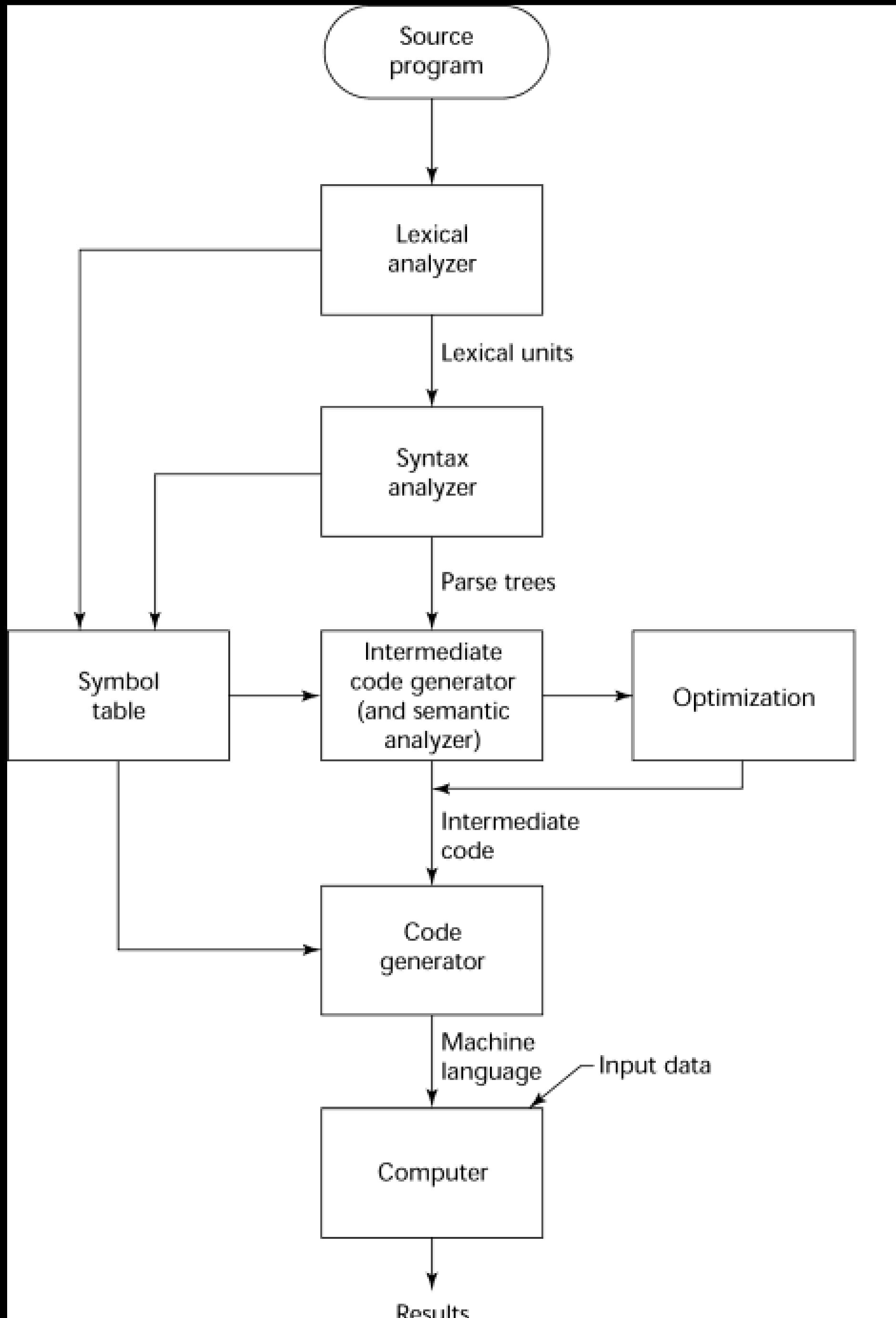



# The translation process



# 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.



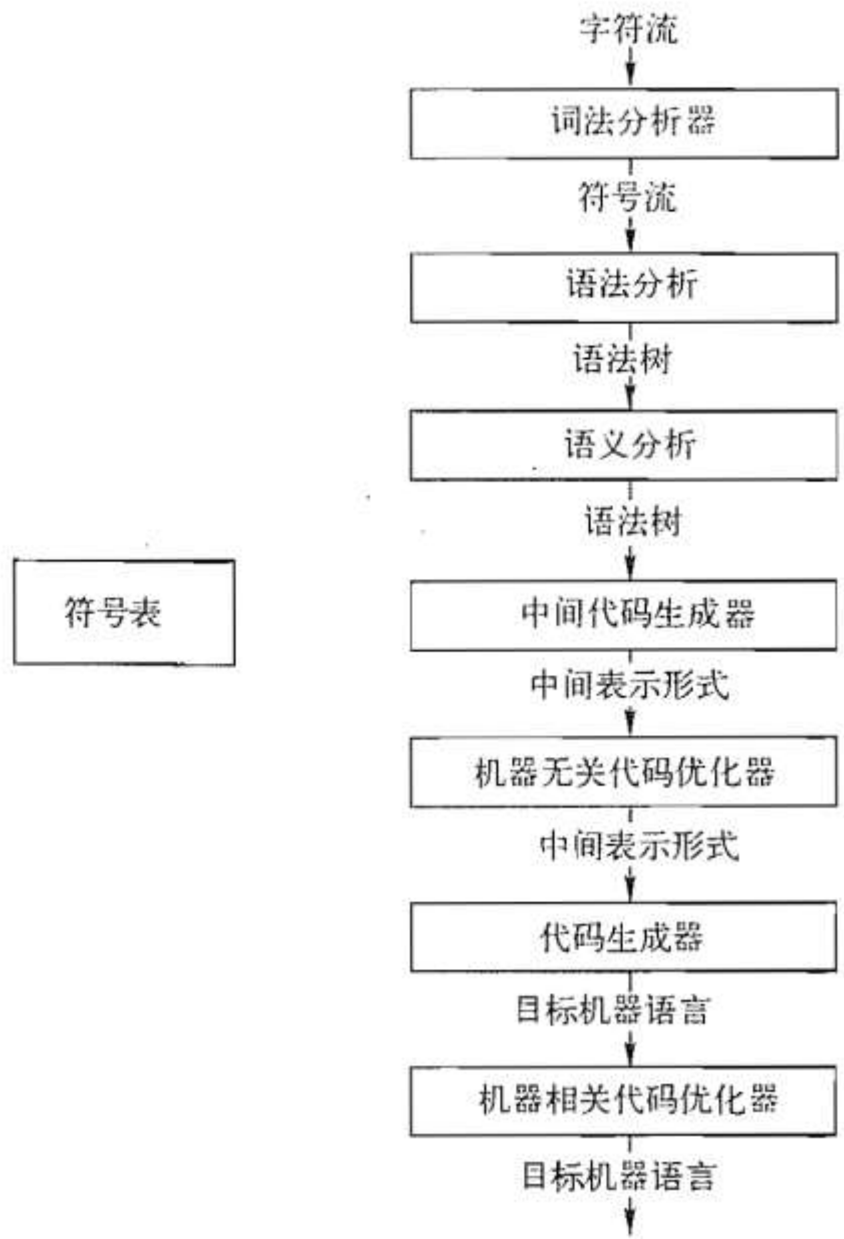



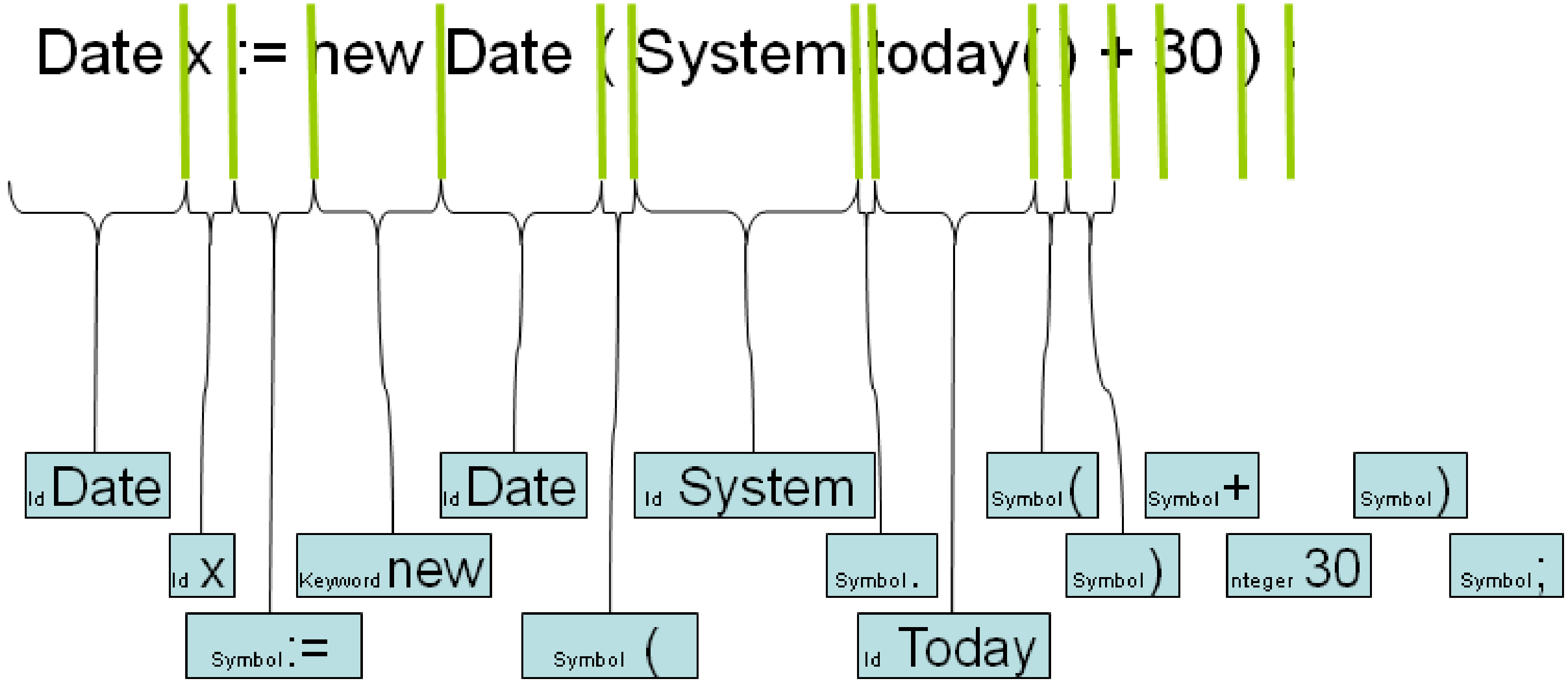

图 1-6　一个编译器的各个步骤

# 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*

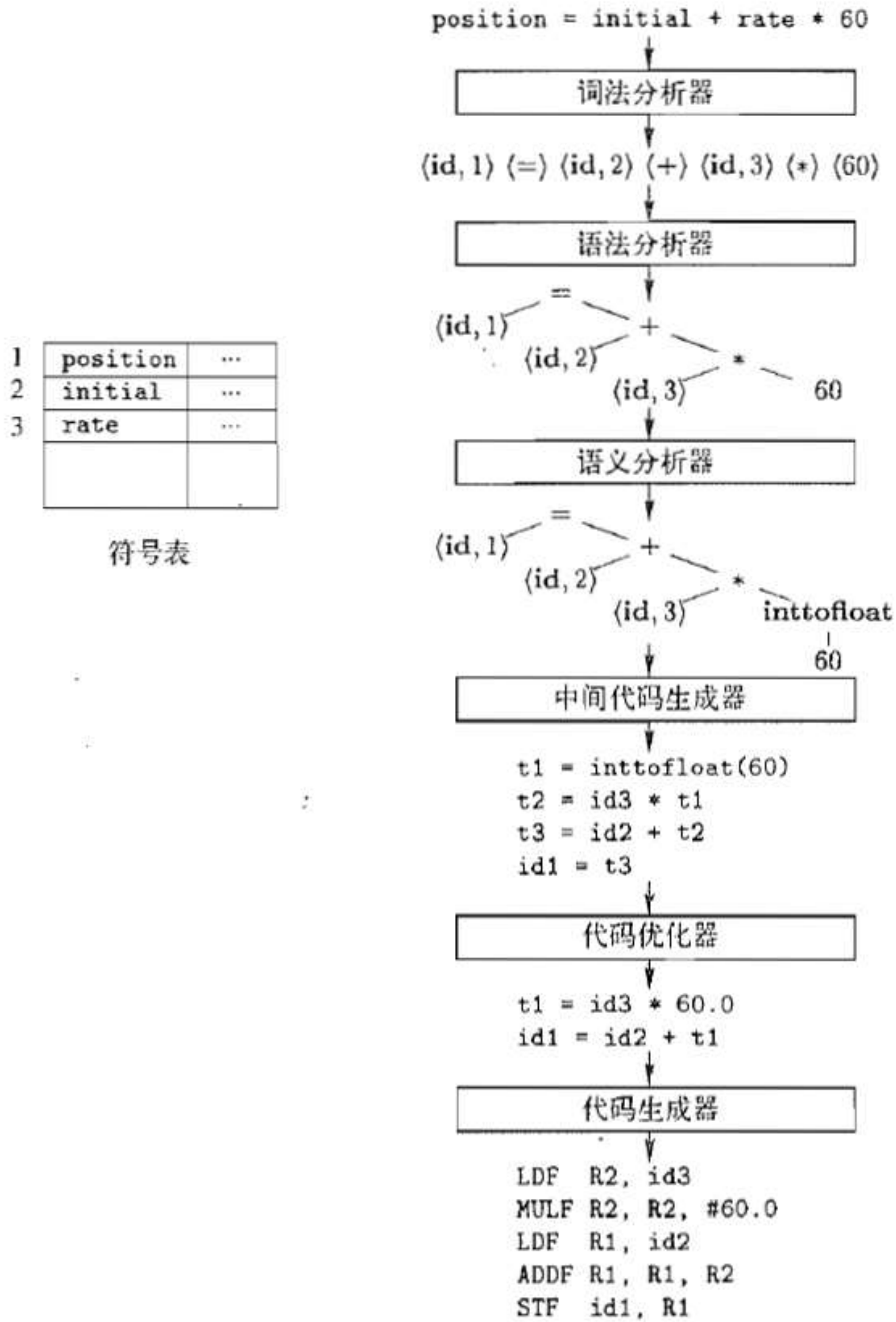


- 词法分析
  - 编译器的第一个步骤称为词法分析或扫描。词法分析器读入组成源程序的字符流，并将其组成有意义的词素的序列。形如<token-name, attribute-value>这样的词法单元。（token-name是由语法分析使用的抽象符号，attribute-value是指向符号表中关于这个词法单元的条目，符号表条目的信息会被语义分析和代码生成步骤使用）

- 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） | |

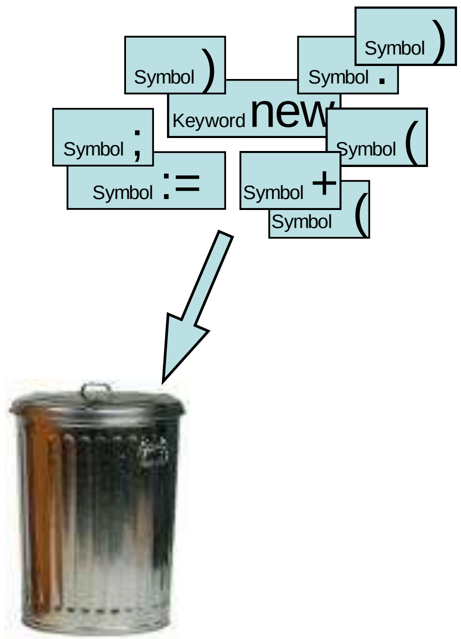
- **经过词**法分析之后，**赋值语**句的**词**法**单**元序列：<id, 1>  < = > <id, 2> < + > <id, 3> < * > <60>

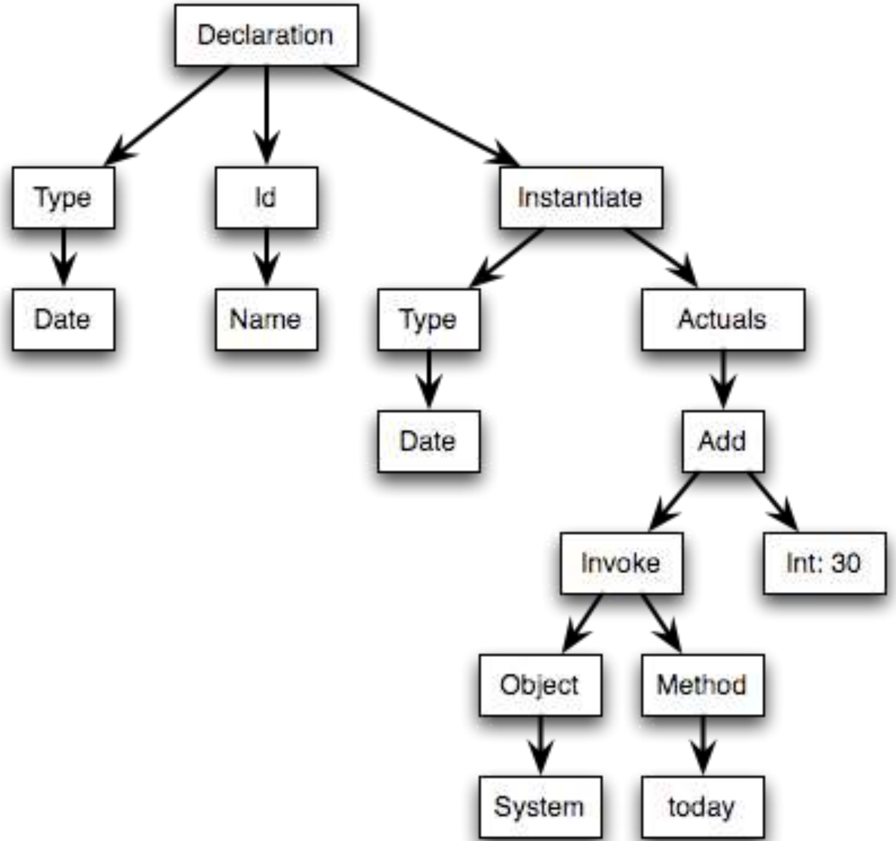

position = initial + rate * 60
词法分析器
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
语法分析器
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
60
语义分析器
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
inttofloat
60
中间代码生成器
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
代码优化器
t1 = id3 * 60.0
id1 = id2 + t1
代码生成器
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
1 position ...
2 initial ...
3 rate ...
符号表

# Major Phases of a Compiler

- Parse （语法分析）
  - Analyze phrase structure and apply semantic actions, usually to build an abstract syntax tree

- **Syntax Analysis (parsing语法分析)**
  - Organize *tokens* in *sentences* based on *grammar*

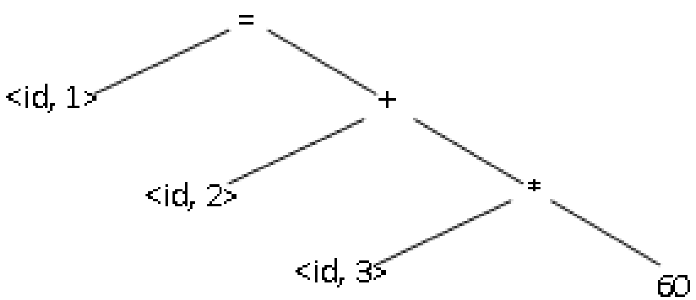


- 语法分析
- 编译的第2个步骤称为语法分析或解析。语法分析器使用由词法分析器生成的各词法单元的第一个分量来创建树形的中间表示。该中间表示给出了词法分析产生的词法单元的语法结构。常用的表示方法是语法树，树中每个内部节点表示一个运算，而该节点的子节点表示运算的分量
- 赋值语句表示成语法树：

position = initial + rate * 60

<id, 1>  < = > <id, 2> < + > <id, 3> < * > <60>



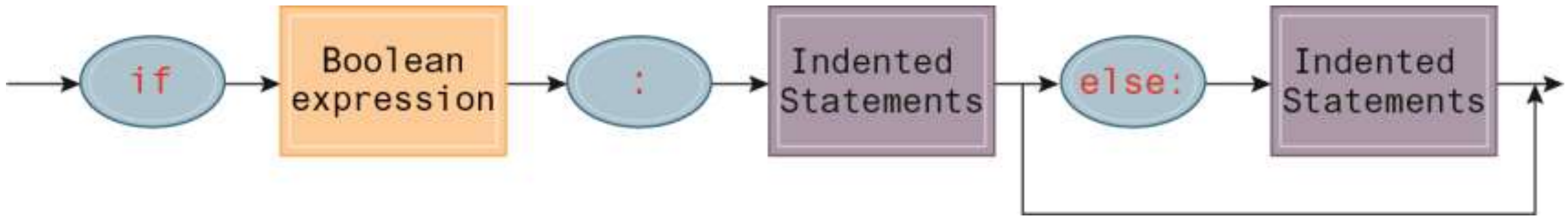
# Syntax Errors

- All programming languages are picky about syntax
- Try this:
  - `1 + 2) + 3`
- Syntax errors can be identified using automated code checking (color coding) and by error messages

# Runtime errors

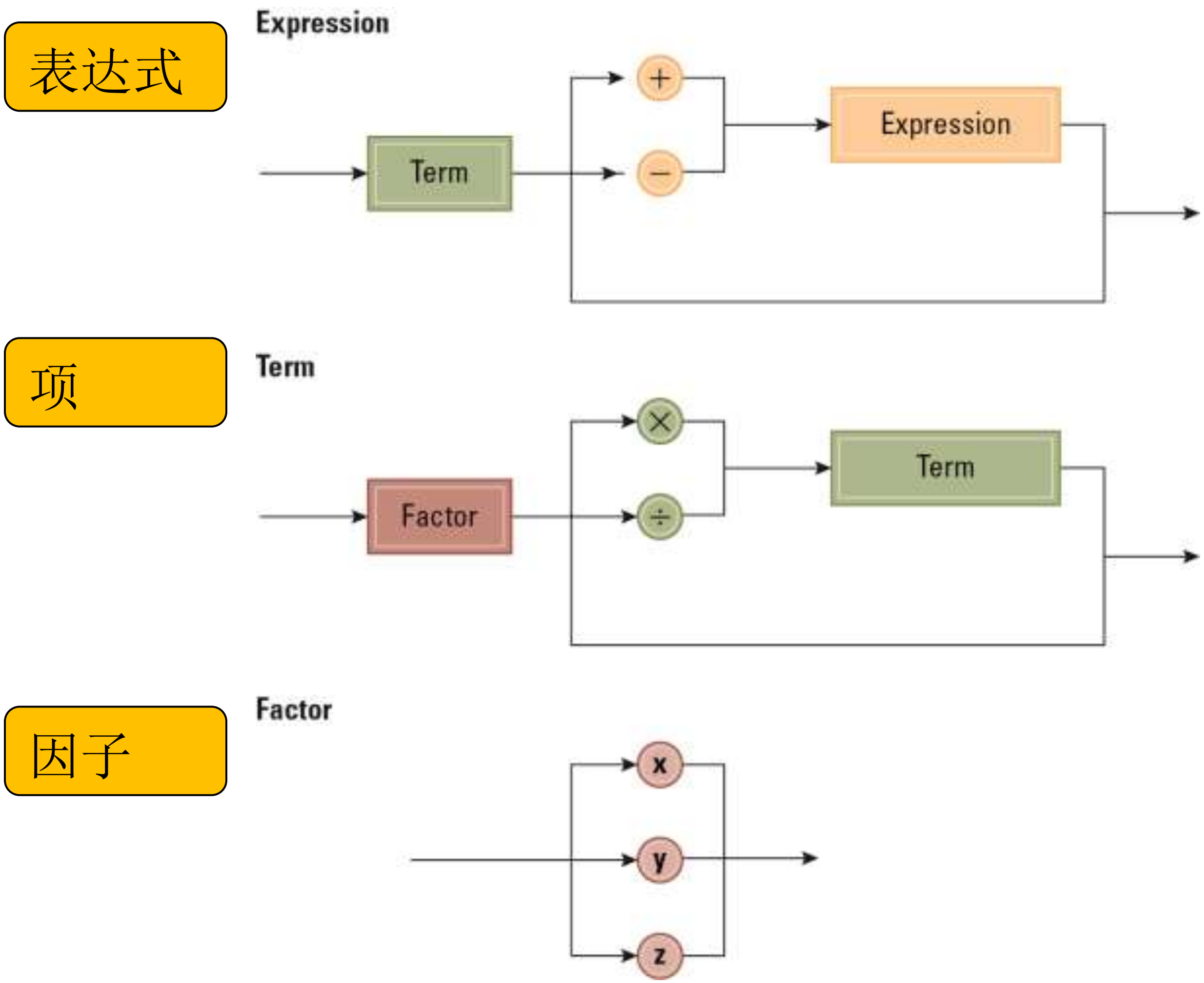
- Errors that occur when program is running
  - Also called “exceptions” 异常
  - Ex: runtime.py
- Identified by testing the program (this includes using test modules)

# Figure 6.14 A syntax diagram of Python's if-then-else statement

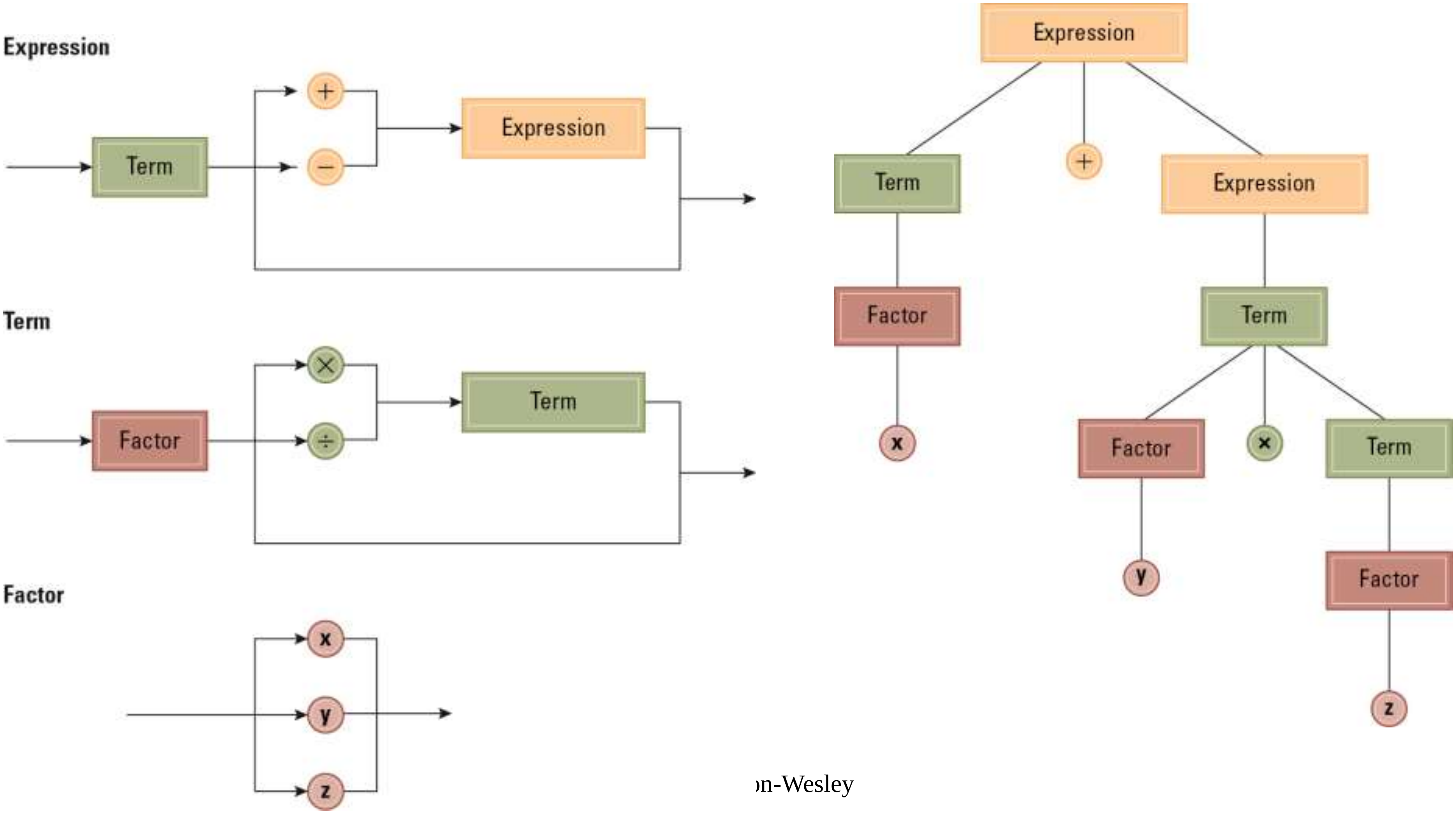



椭圆：终结符
矩形：非终结符（需进一步描述）

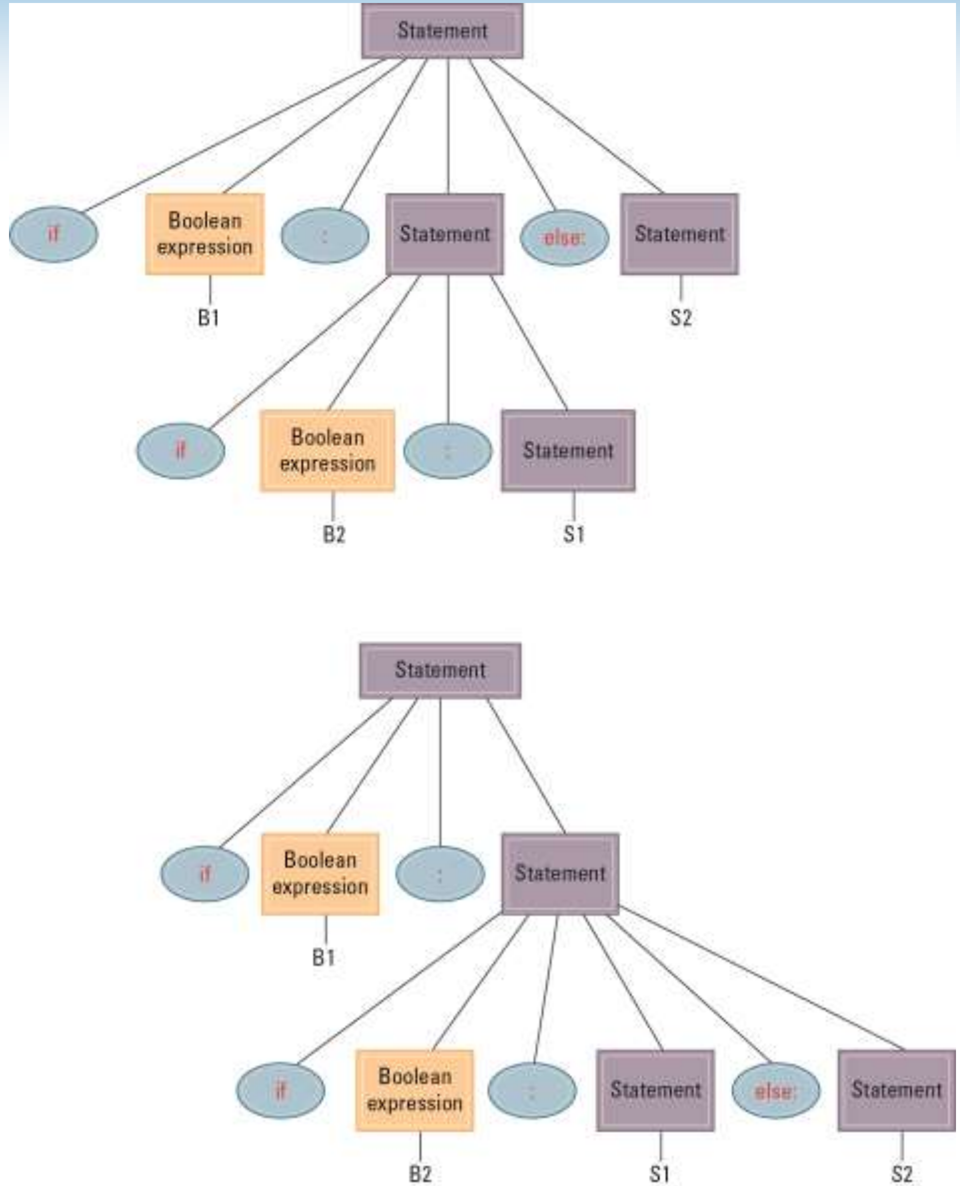
# Figure 6.15 Syntax diagrams describing the structure of a simple algebraic expression



# Figure 6.16 The parse tree for the string x + y * 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

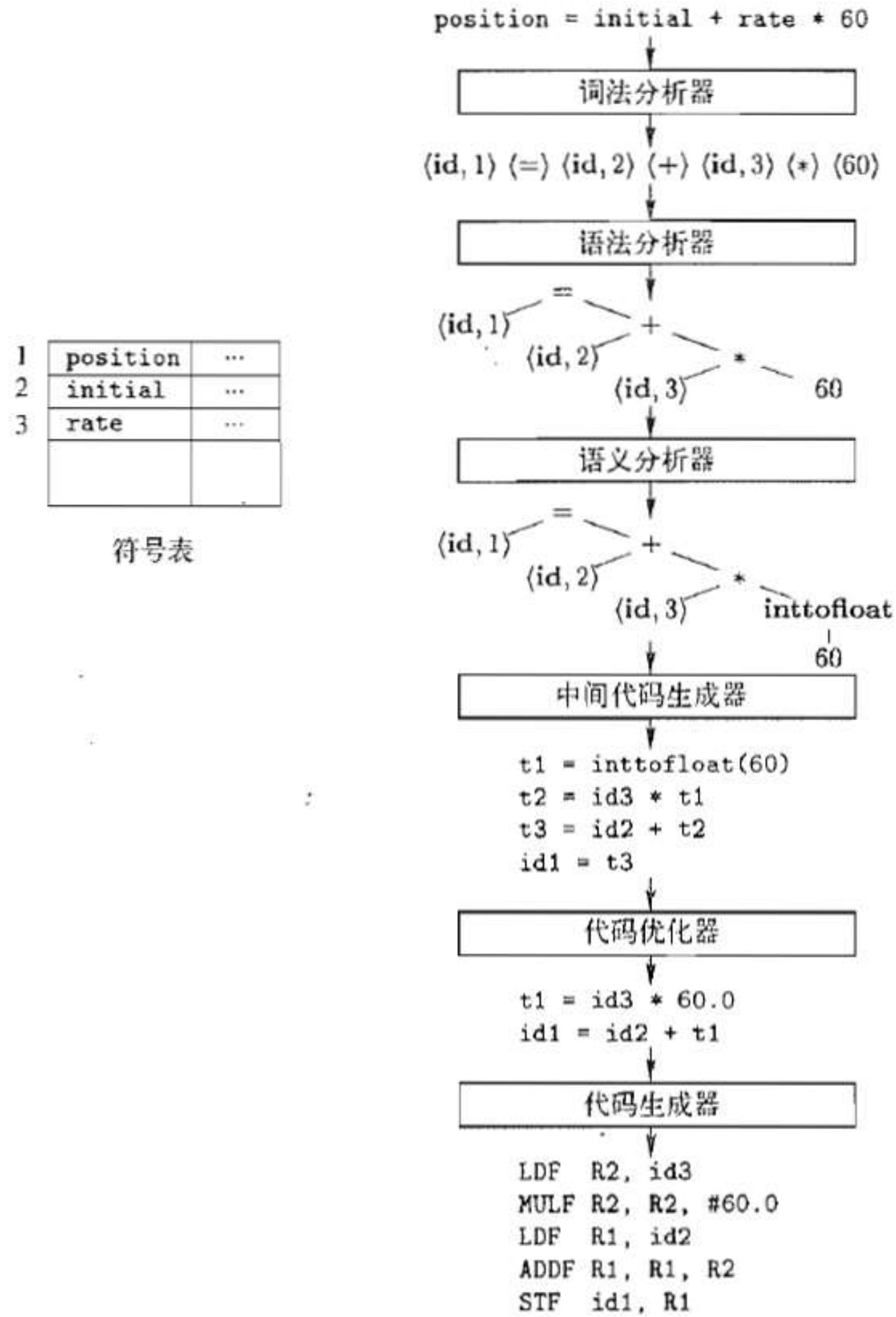


# Major Phases of a Compiler

- Semantic analysis （语义分析）
  - Determine what each phrase means, connect variable name to definition (typically with symbol tables), check types

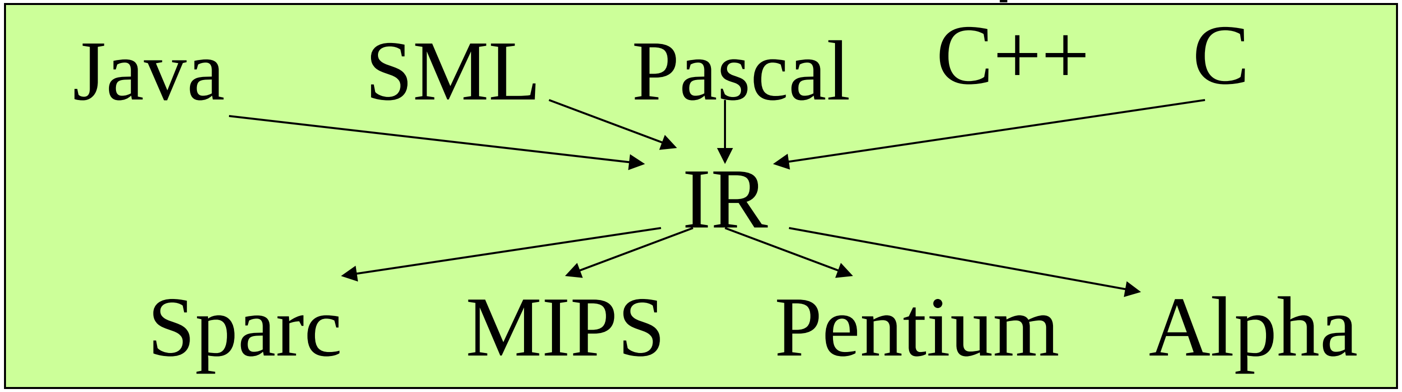
- 语义分析器使用**语法树**和符号表中的信息来**检查**源程序是否和**语**言定**义**的**语义**一致。它同**时**收集类型信息，并存放在**语**法**树**或符号表中，以便在中**间**代**码**生成**过**程使用。
- **语义**分析的一个重要部分就是类型**检查**。比如很多**语**言要求数**组**下**标**必**须为**整数，如果使用浮点数作**为**下**标**，**编译**器就必**须报错**。再比如，很多**语**言允**许**某些类型**转换**，称**为**自**动**类型**转换**。

- 语义分析
- 自动类型转换
- 假设position, initial和rate已经被声明为浮点型，而词素60是一个整数。语义分析器输出中有一个inttofloat的额外节点，明确的把60转换为一个浮点数



# Major Phases of a Compiler

- Translate to intermediate representation




- Instruction selection
- Optimize
- Emit final machine code

# 中间代码生成

在源程序翻译成目标代码的过程中，一个编译器可能构造出一个或多个中间表示。这些中间表示可以有多种形式，语法树是一种中间表示形式

很多编译器生成一个明确的低级的或类机器语言的中间表示。该中间表示有两个重要的性质：1.易于生成；2.能够轻松地翻译为目标机器上的语言。

## 代码优化

代码优化试图改进中间代码，以便生成更好的目标代码。即更快（省时），更短（省空间）或能耗更低。

## 代码生成

代码生成以中间表示形式作为输入，并把它映射为目标语言。如果目标语言是机器代码，则必须为每个变量选择寄存器或内存位置，中间指令则被翻译为能够完成相同任务的机器指令序列。

代码生成的一个至关重要的方面是合理分配寄存器以存放变量的值

# Example of Intermediate Representation

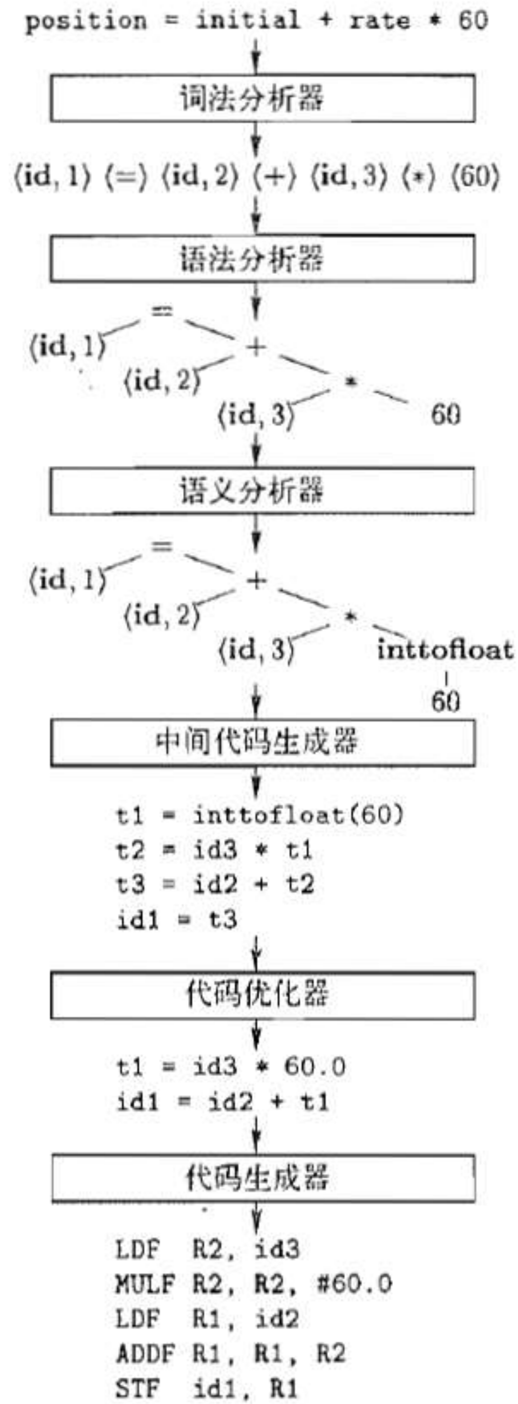
- Program code: X = Y + Z + W
  - tmp = Y + Z
  - X = tmp + W
  - Simpler language with no compound arithmetic expressions

# Example of Optimization

Program code: $X = Y + Z + W$

| | |
|---|---|
| • Load reg1 with Y<br>• Load reg2 with Z<br>• Add reg1 and reg2, saving to reg1<br>• Store reg1 to tmp ** | • Load reg1 with tmp **<br>• Load reg2 with W<br>• Add reg1 and reg2, saving to reg1<br>• Store reg1 to X |

Eliminate two steps marked **

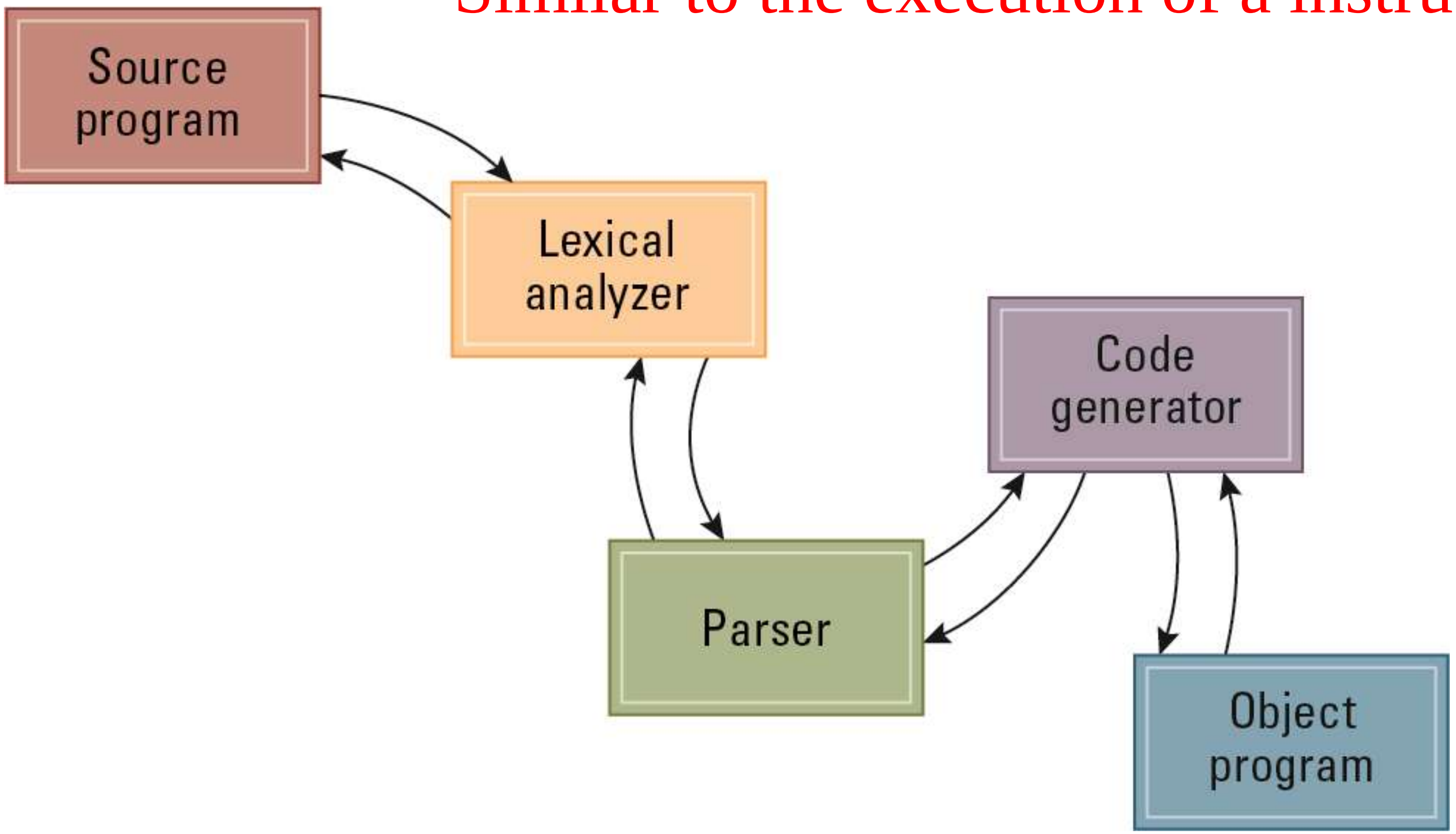



| | | |
|---|---|---|
| 1 | position | ··· |
| 2 | initial | ··· |
| 3 | rate | ··· |
| | | |

符号表

# Figure 6.18 An object-oriented approach to the translation process

Similar to the execution of a instruction



# 6.5 Object-Oriented Programming

- **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.

# 类和对象

## (1)类

对同种客观事物的抽象,包含特征(属性)描述和行为(方法).

将反映类的属性、方法、事件封装在一起，构成了面向对象编程的基本元素。

- **属性**：对象的外观特征数据
- **方法**：对象的行为、能力
- **事件**：外界发生在对象上的事情

人类是具体人的抽象
属性：身高、体重等
行为：哭、笑、打电脑等
事件：下雨、铃声

# (2) 对象

是类的实例化。

例如，张三、李四就是人类的实例化，每个人有各自不同的属性值和方法。

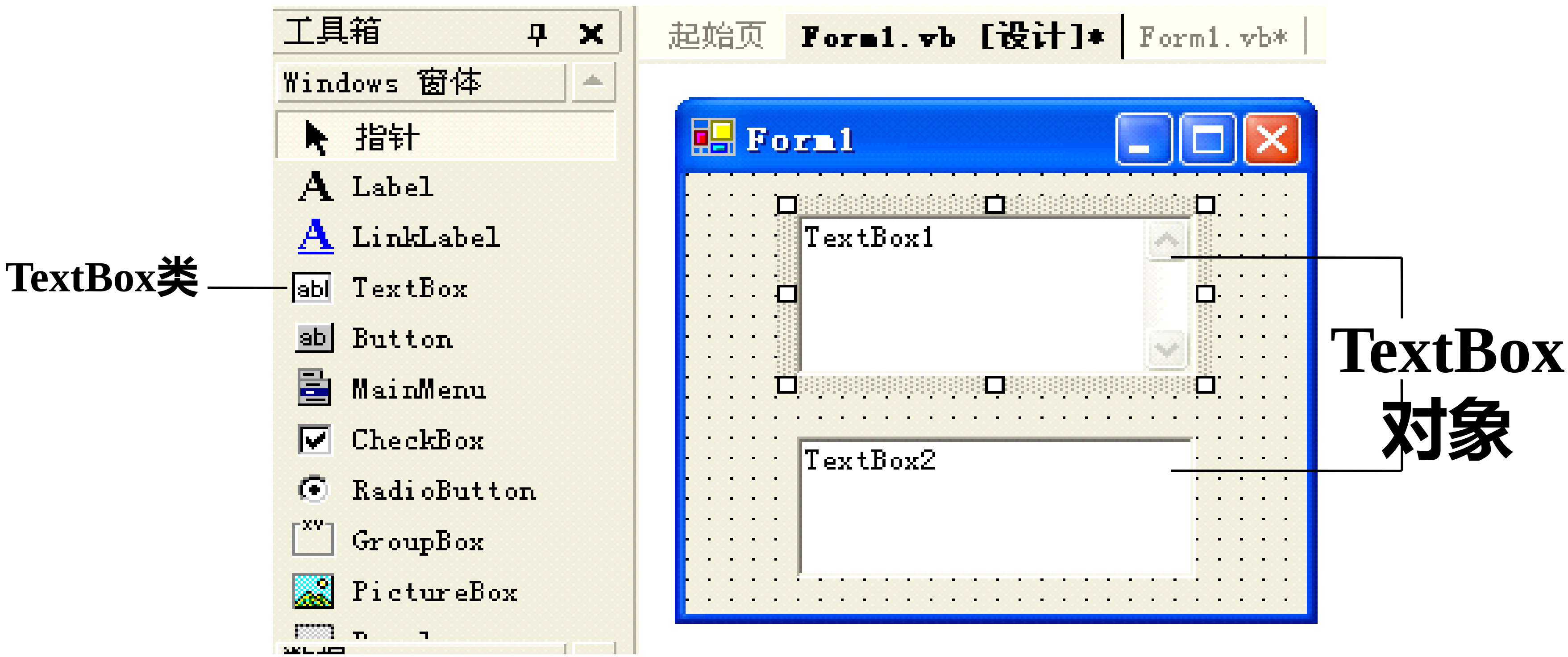

实例化


月饼模型（类）　　　　月饼（对象）

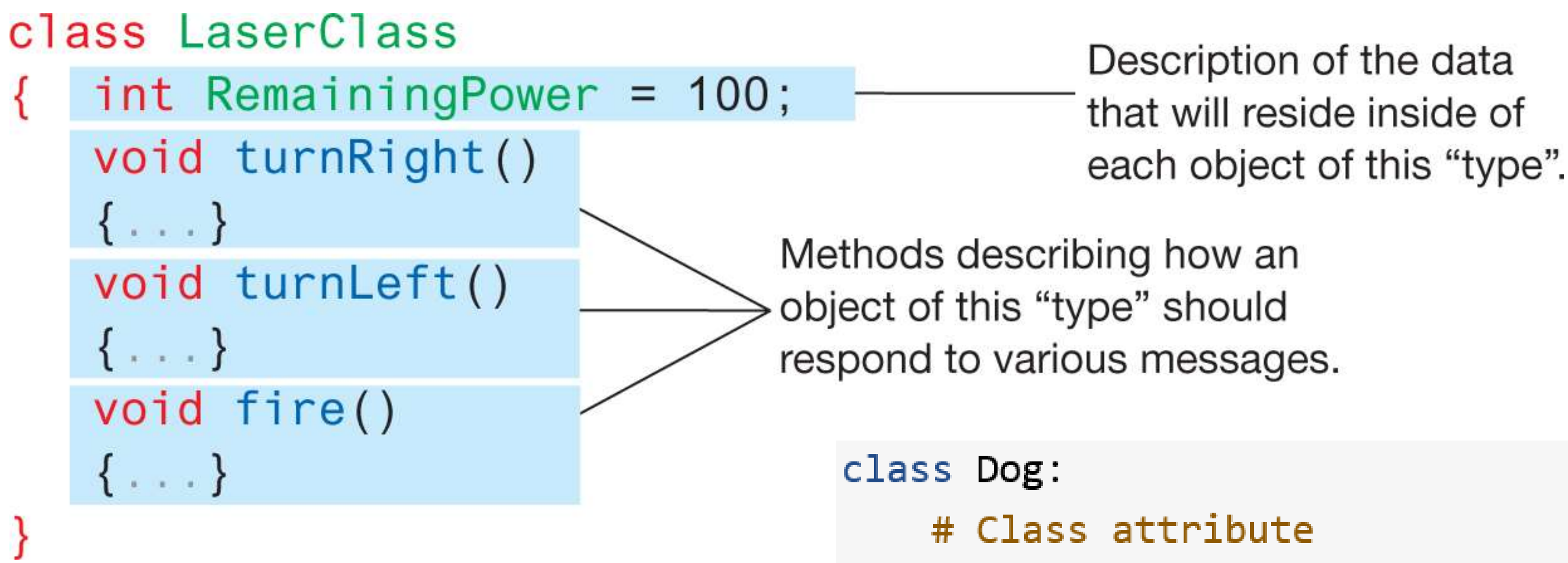
# VB.NET中的可视化类和对象

**例如**：工具箱内的**TextBox**是类(它确定了**TextBox**的属性、方法和事件)

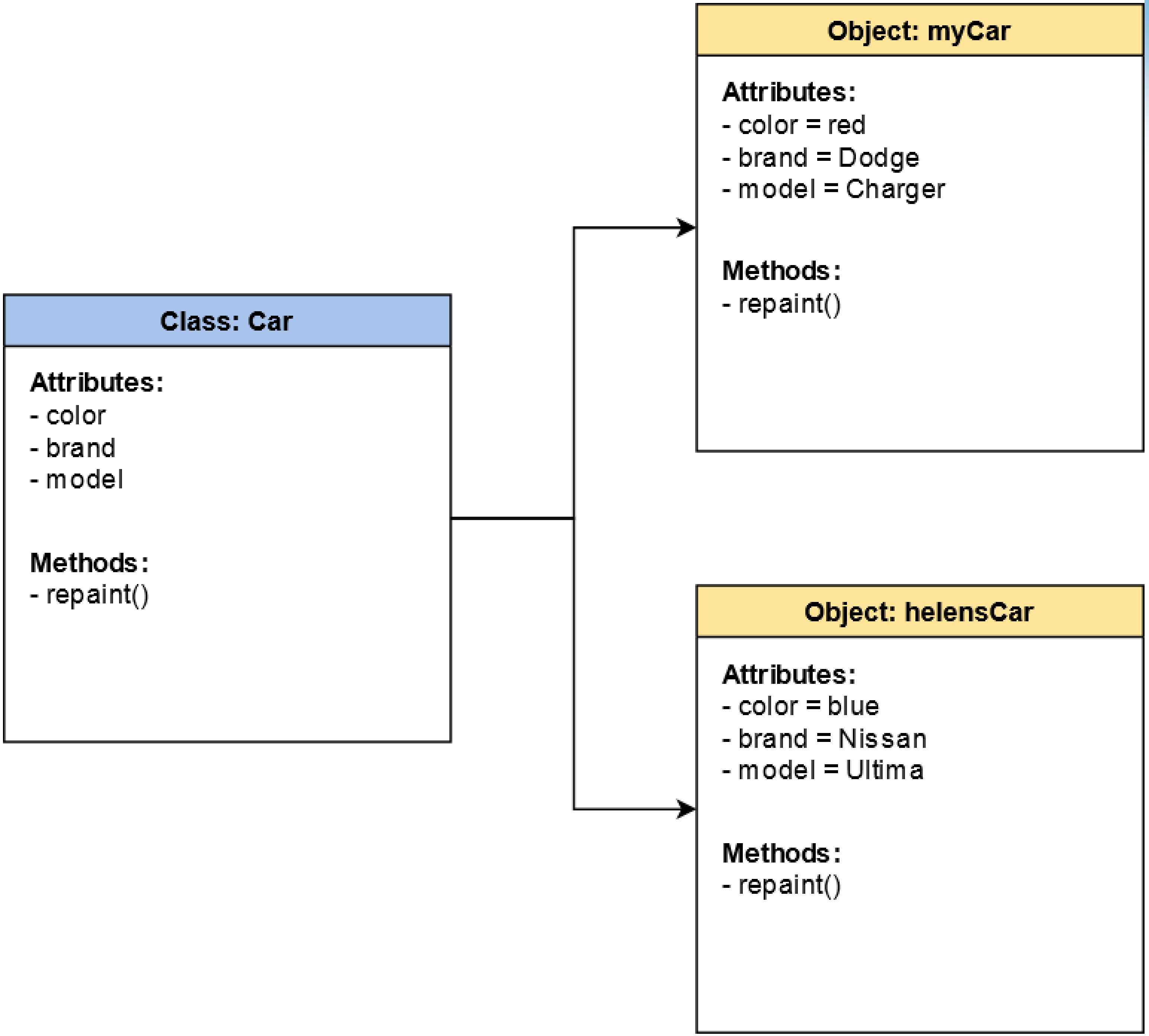
窗体上显示的是两个**TextBox**对象



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




```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```


Class: Car
Attributes:
- color
- brand
- model
Methods:
- repaint()
Object: myCar
Attributes:
- color = red
- brand = Dodge
- model = Charger
Methods:
- repaint()
Object: helensCar
Attributes:
- color = blue
- brand = Nissan
- model = Ultima
Methods:
- repaint()

# 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

```
class LaserClass
{ int RemainingPower;
  LaserClass(InitialPower)
  { RemainingPower = InitialPower;
  }
  void turnRight()
  {. . .}
  void turnLeft()
  {. . .}
  void fire()
  {. . .}
}
```

Constructor assigns a value to RemainingPower when an object is created.

```python
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

Python

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

# Python

```python
>>> miles = Dog("Miles", 4)

>>> miles.description()
'Miles is 4 years old'

>>> miles.speak("Woof Woof")
'Miles says Woof Woof'

>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

# Additional Object-oriented Concepts

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

多态就是同一操作（方法）作用于不同的对象时，可以有不同的解释，产生不同的执行结果

# Inheritance

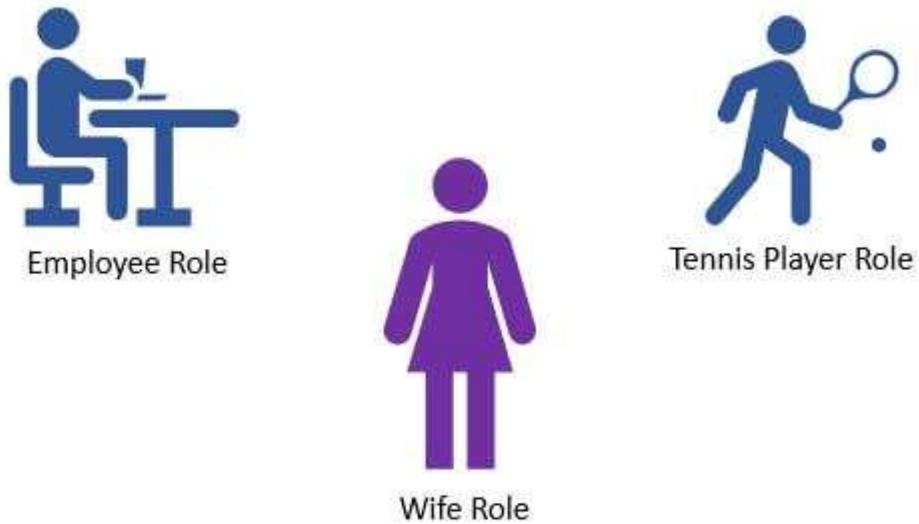
```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

```python
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")
```

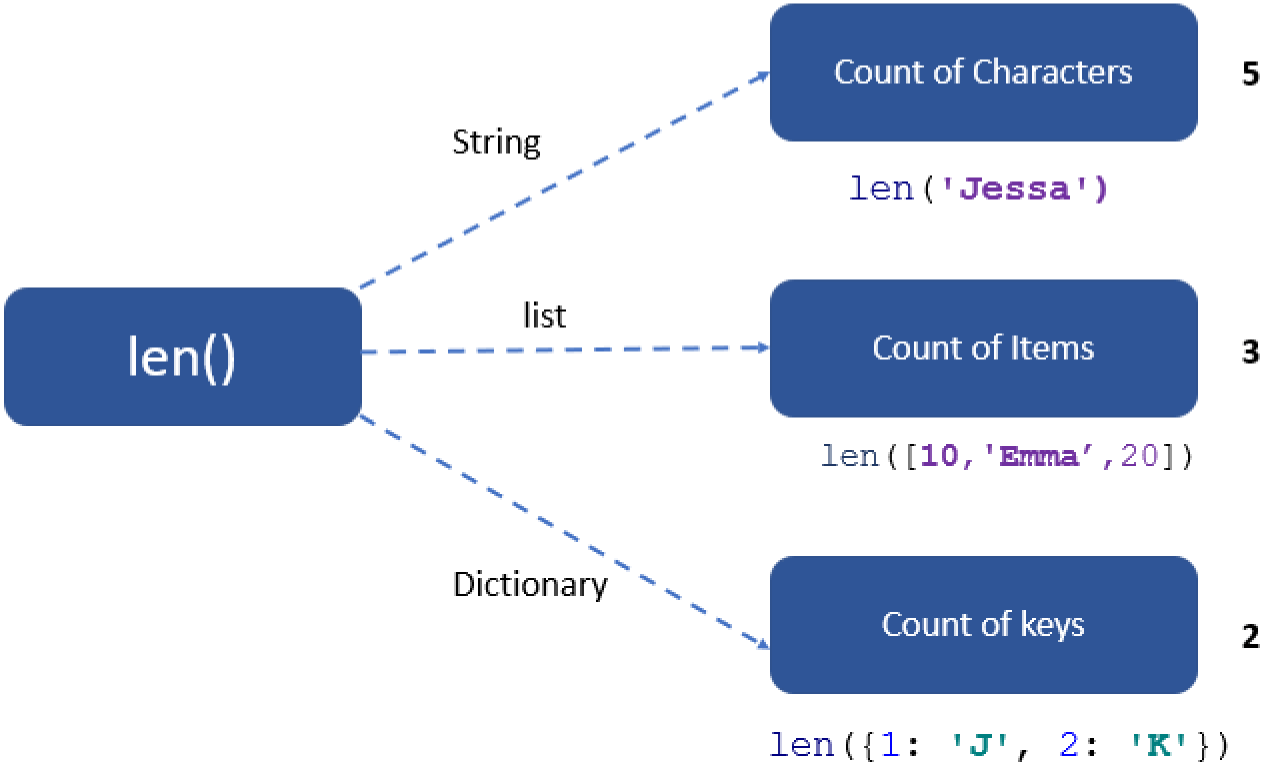
```python
# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

# Polymorphism多态

- In polymorphism, a method can process objects differently depending on the class type or data type

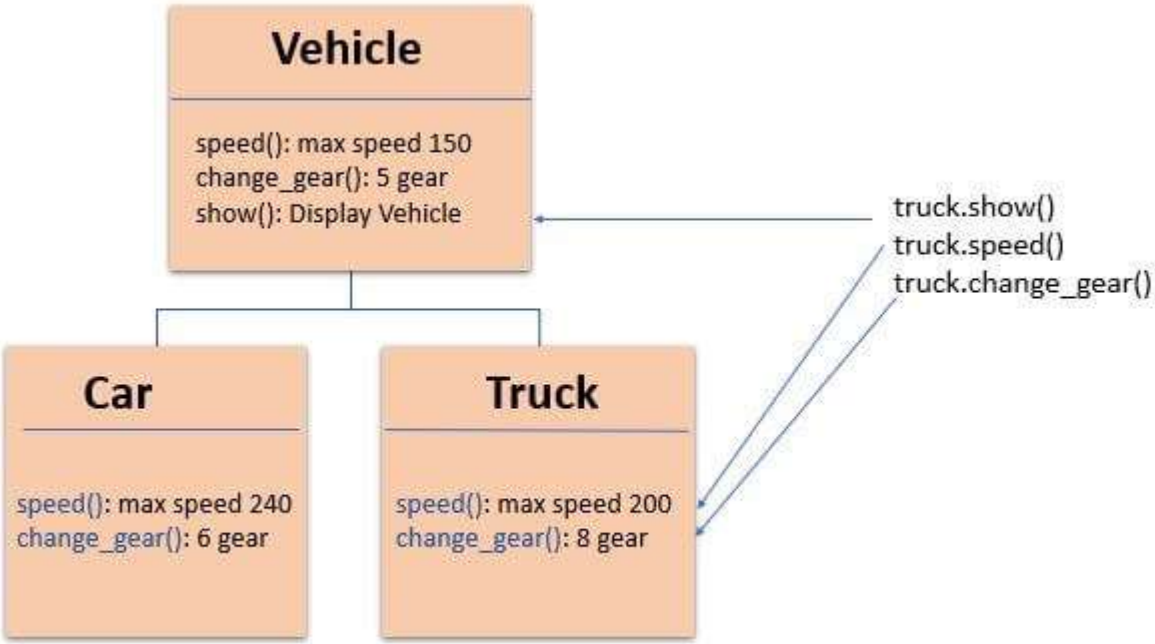



Jessa takes different forms as per the situation




Polymorphic len() function

- Using method overriding polymorphism allows us to defines methods in the child class that have the same name as the methods in the parent class.
- This process of re-implementing the inherited method in the child class is known as Method Overriding.




Method overridden in Car and Truck class

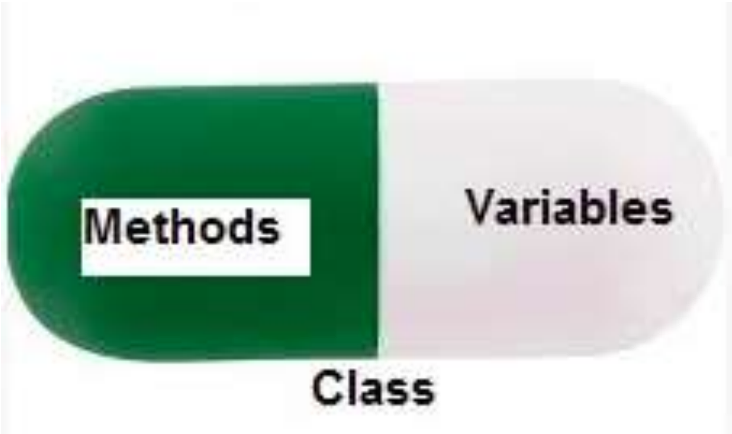
```python
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()
```

```python
#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

```
Parrot can fly
Penguin can't fly
```

# Object Integrity

- **Encapsulation封装:** 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

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 (   )
 {. . .}
}
```

```python
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

we have tried to modify the value of __maxprice outside of the class. However, since __maxprice is a private variable, this modification is not seen on the output.

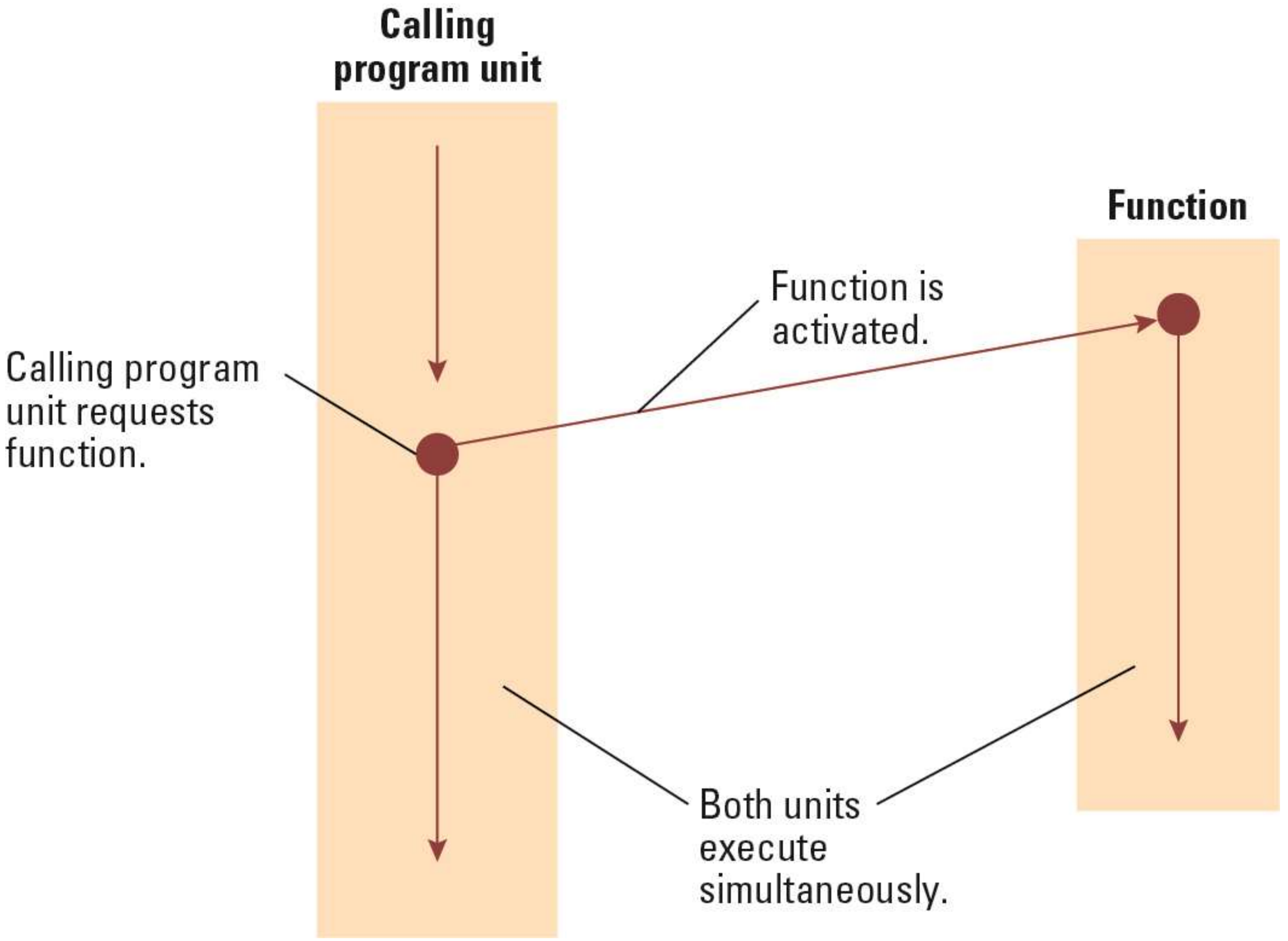
To change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

# 6.6 Programming Concurrent Activities

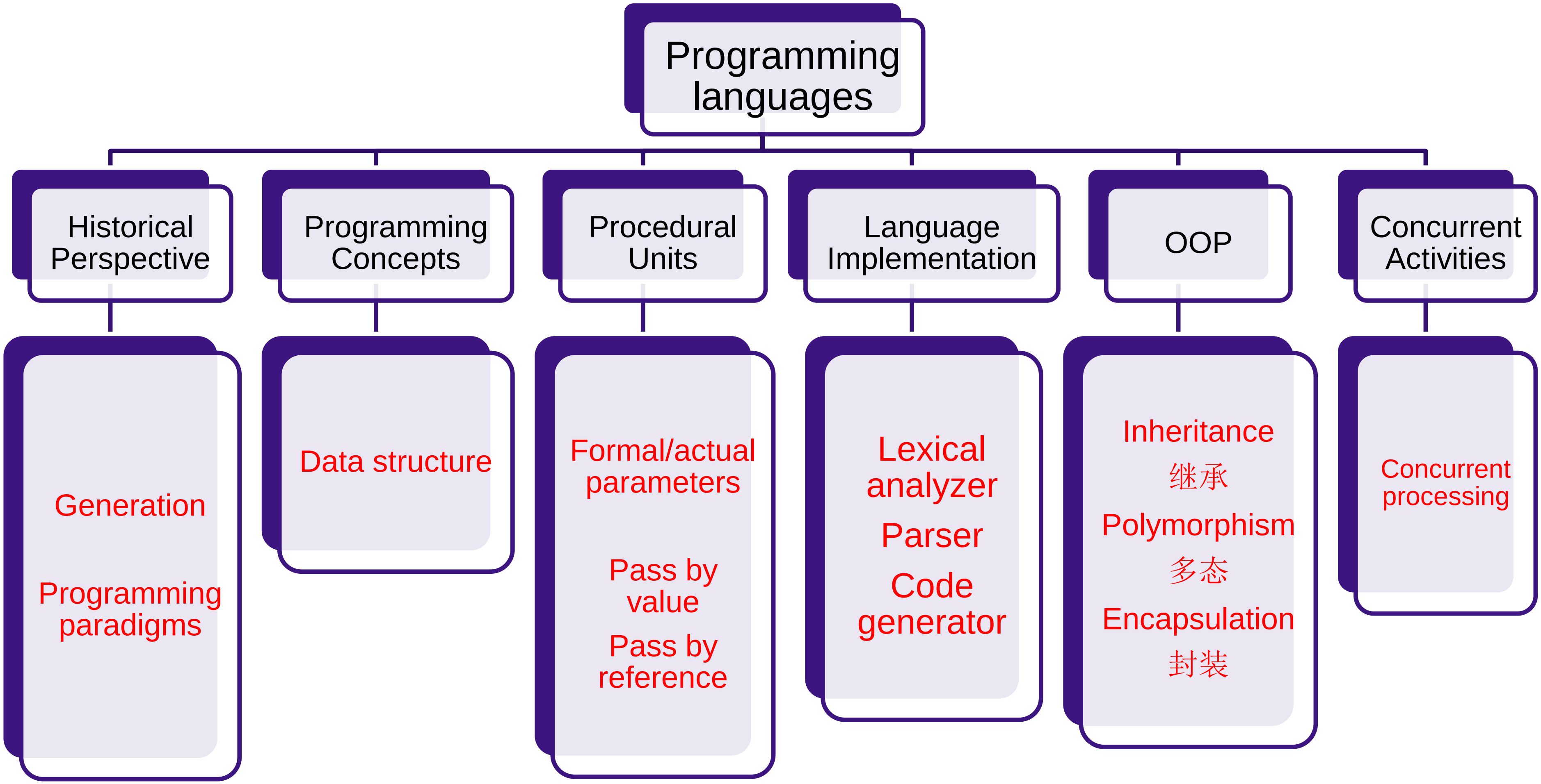
- **Parallel** (or **concurrent**) **processing:** simultaneous execution of multiple processes
  - True concurrent processing requires multiple CPUs
  - Can be simulated using time-sharing with a single CPU

# 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


Programming languages
Historical Perspective
Programming Concepts
Procedural Units
Language Implementation
OOP
Concurrent Activities
Generation
Programming paradigms
Data structure
Formal/actual parameters
Pass by value
Pass by reference
Lexical analyzer
Parser
Code generator
Inheritance
继承
Polymorphism
多态
Encapsulation
封装
Concurrent processing