Programming Languages

Introduction to Computer

Yu-Ting Wu

(with most slides borrowed from Prof. Tian-Li Yu)

Outline

- Historical perspective
- Traditional programming concepts
- Procedural units
- Language translation process
- Object-oriented programming
- Programming concurrent activities

Outline

- Historical perspective
- Traditional programming concepts
- Procedural units
- Language translation process
- Object-oriented programming
- Programming concurrent activities

Programming Language Generations

1st	2nd	3rd	4th ?
Machine instructions	Assembly	Fortran Cobol Basic C/C++ Java	SQL SAS

From Machine Instructions to Assembly

1st Machine instructions		2nd Assembly
156C	→	LD R5, Price
166D	→	LD R6, ShippingCharge
5056	→	ADDI R0, R5, R6
306E	→	ST R0, TotalCost
C000	→	HTL

6E = 6C + 6D

TotalCost = Price + ShippingCharge

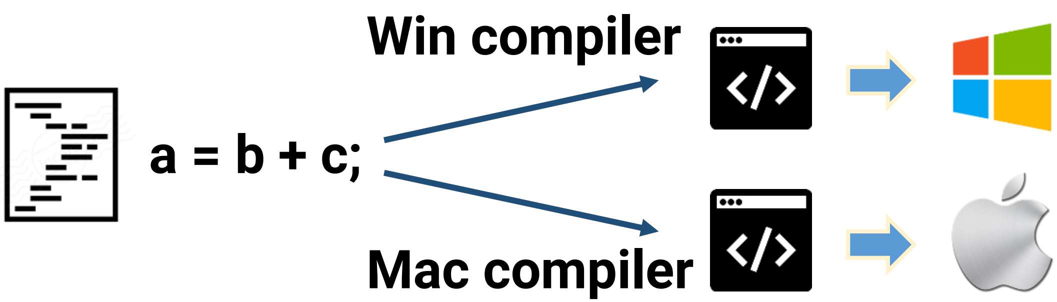
- **Mnemonic** names for op-codes
- **Program variables** or **identifiers**: descriptive names for memory locations, chosen by the programmer

Assembly Language Characteristics

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

3rd Generation (High-level) Language

- Use high-level primitives
  - E.g., if-then, do-while
- Each primitive corresponds to a sequence of machine language instructions
- **Machine independent (mostly)**
- Converted to machine language by a program called a **compiler (or interpreter)**



Programming Languages and Issues

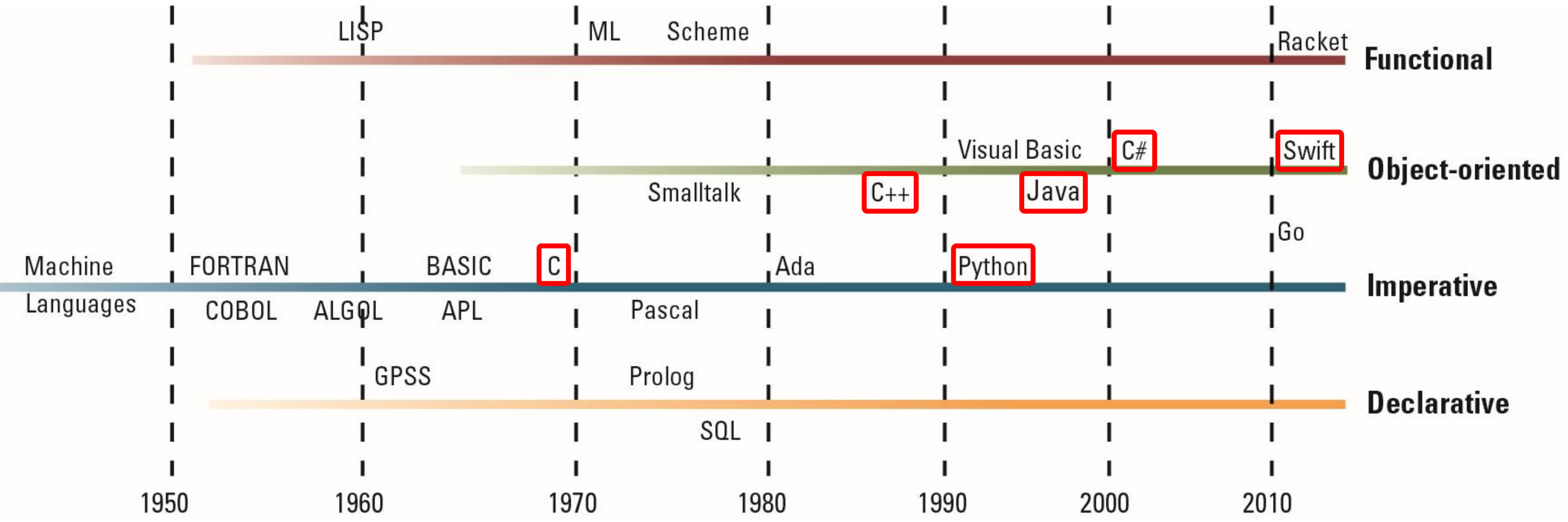
- Natural v.s. Formal languages
- Formal language
  - Use formal grammar

    Expression → Term | Term + Expression | Term – Expression
    Term → Factor | Factor * Term | Factor / Term
    Factor → x | y | z

    x + y * z

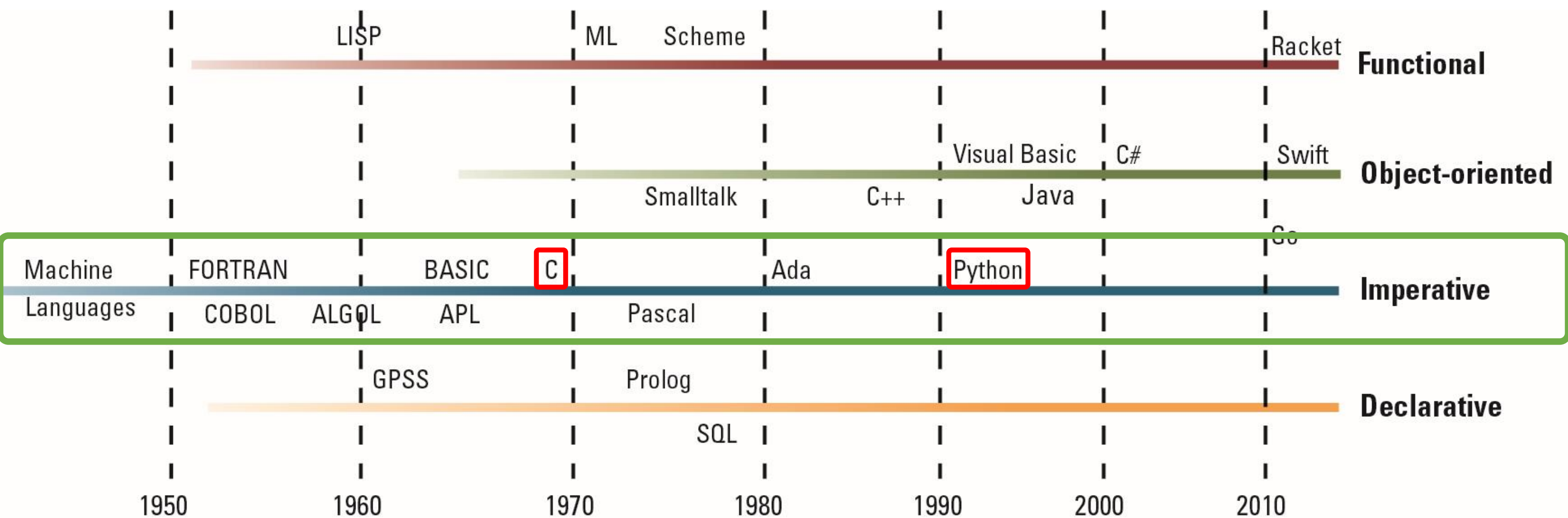
  - Will be introduced later
- **Portability**
  - Theoretically: same source code, different compilers
  - Reality: minor modifications

Programming Language Paradigms



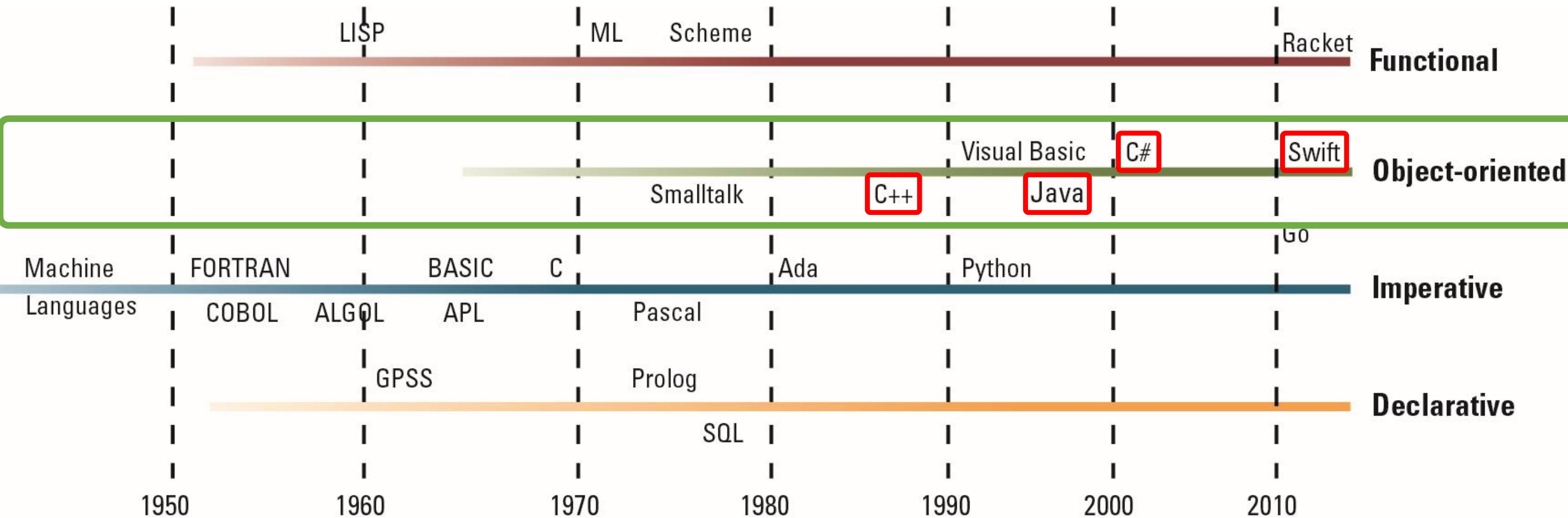
Imperative Paradigms

- Procedural
- Approach a problem by **finding an algorithm** to solve the problem



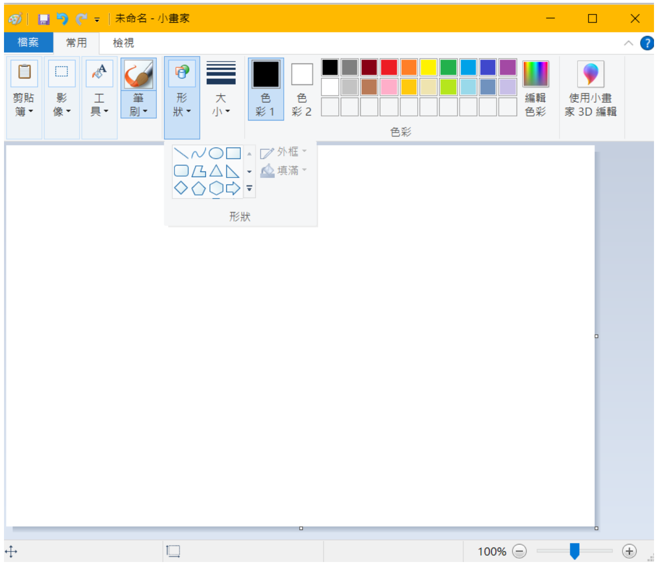
Object-oriented Paradigms

- Implements **objects and their associated procedures** within the programming context to create software programs
- Information hiding, inheritance, polymorphism



Object-oriented Paradigms (cont.)

- Example: a painter program



Canvas

Shape
Draw();

Eraser

Circle
Draw();

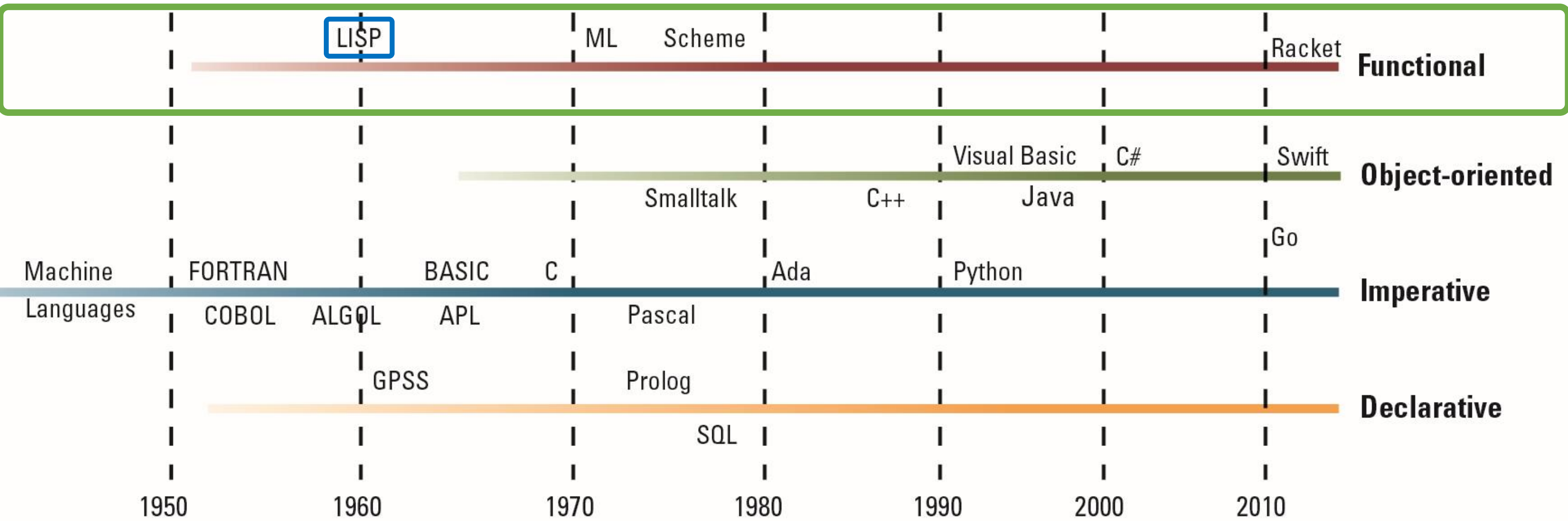
Rectangle
Draw();

Line
Draw();

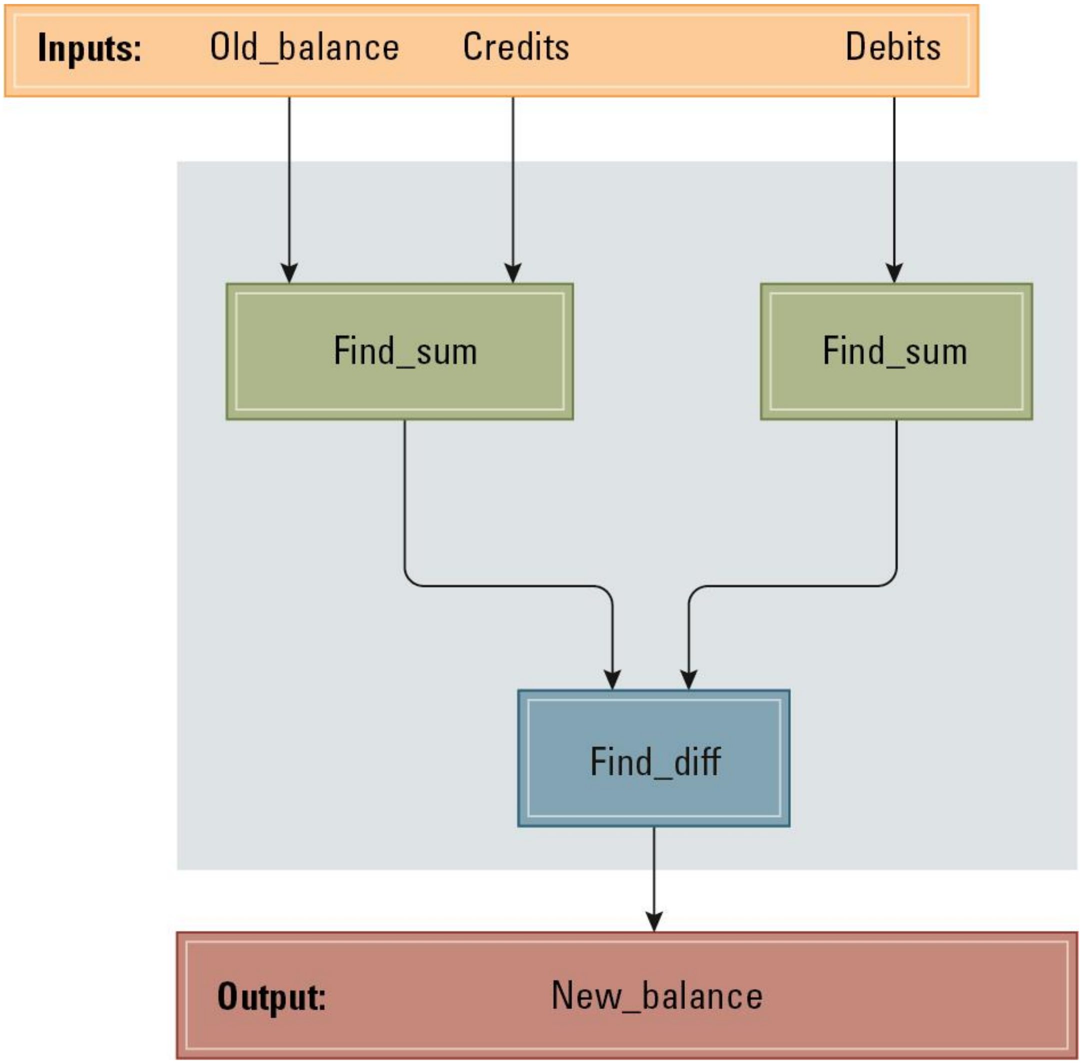
```
Shape* shapeList[10];
foreach shape in shapeList
    shape->Draw();
```

Functional Paradigms

- Treat the entire program as a function
- A program consists of sub-problems that are handled by sub-functions



Functional Paradigm (cont.)



Functional v.s. Imperative

Temp_balance ← Old_balance + Credit

Total_debits ← sum of all Debits

Balance ← Temp_balance – Total_debits

LISP (f x y)

(**Find_diff** (**Find_sum** Old_balance Credits) (**Find_sum** Debits))

Sum ← sum of all Numbers

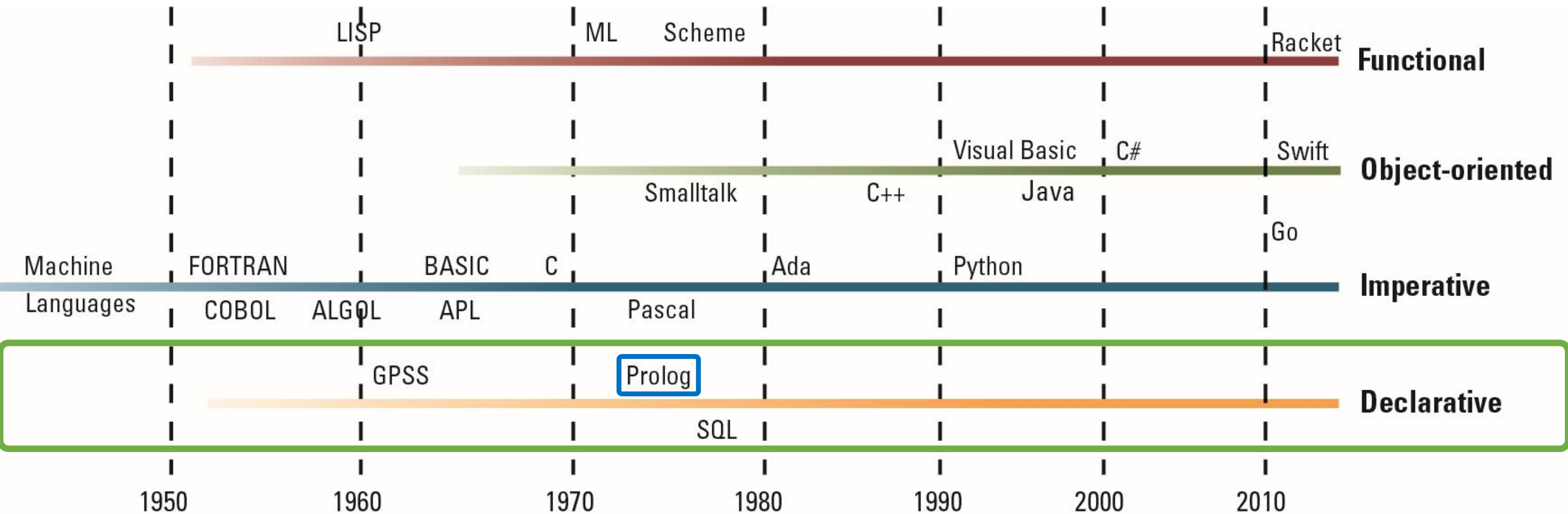
Count ← # of Numbers

Average ← Sum / Count

(**Find_average** (**Find_sum** Numbers) (**Find_count** Numbers))

Declarative Paradigms

- Implemented as a **general problem solver**
- Approach a problem by **finding a formal description of the problem**
  - E.g., define ***factorial*** by 0! = 1 and n! = n * (n-1)!



Outline

- Historical perspective
- Traditional programming concepts
- Procedural units
- Language translation process
- Object-oriented programming
- Programming concurrent activities

Traditional Programming Concepts

- Variables and data types
- Data structure
- Constants and literals
- Assignments and operators
- Control
- Comments

Variables and Data Types

- **Integer**: whole numbers
- **Floating-point (Real)**: numbers with fractions
- **Character**: symbols
- **Boolean**: true/false

C/C++, Java

```
int a;
float b;
char c;
bool d;
```

FORTRAN

```
INTEGER a;
REAL b;
BYTE c;
LOGICAL d;
```

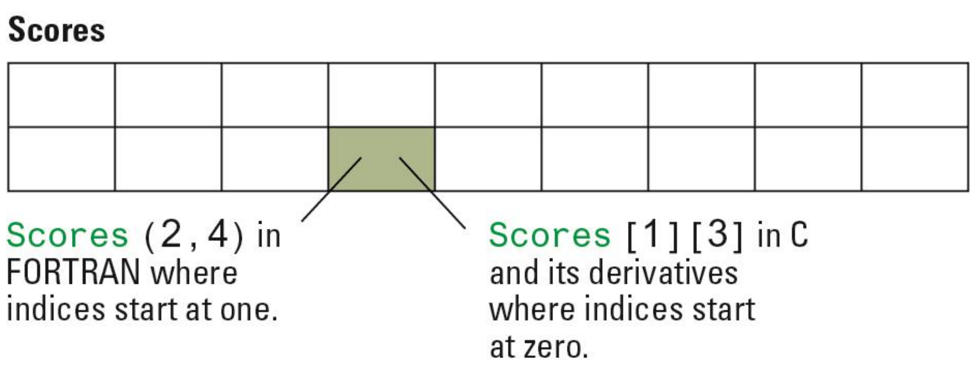
Data Structures

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

C/C++, Java
int a[5][100];

FORTRAN
INTEGER a(5, 100);

- The starting index might differ in different programming languages

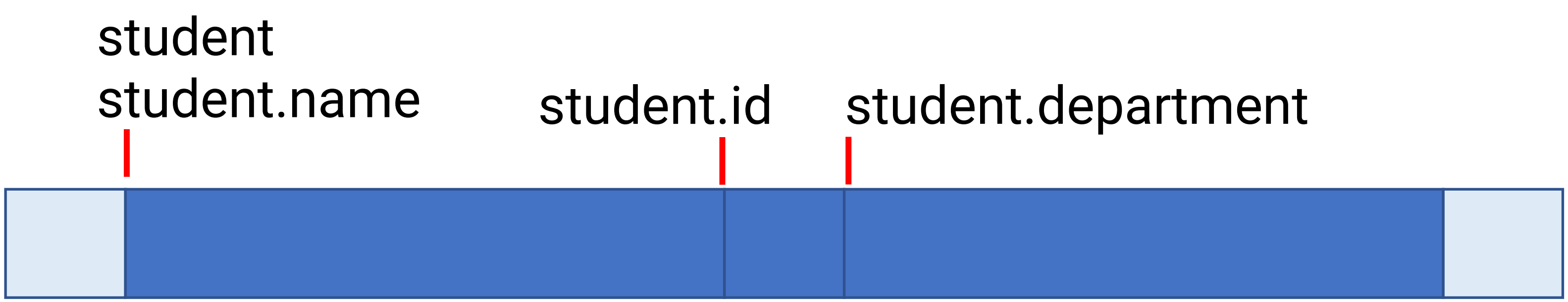


Data Structures (cont.)

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

C/C++

```
struct Student {
	char name[30];
	int id;
	char department[30];
};
```



Literals and Constant

- **Literal**
  - a ← b + $\boxed{\text{100;}}$
- **Constant**
  - Const int a = 100; (C/C++)
  - final int a = 100; (Java)
  - A constant cannot be a **l-value**
    - const int a = 100;
      a = b + c; ✗

Assignment and Operators

- **Assignment**
  - a = b + c; (C/C++/Java)
- **Operators**
  - Operator precedence
    - E.g., int a = 3 + 4 * 5 | 6;
    - https://en.cppreference.com/w/c/language/operator_precedence
  - Operator overloading

```cpp
struct Complex {
    int real;
    int imag;
    Complex operator+ (Complex const& obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
};
```

```cpp
Complex c1, c2;
c1.real = 3;    // c1 = 3 + 4i
c1.imag = 4;
c2.real = 5;    // c2 = 5 + 6i
c2.imag = 6;
Complex c3 = c1 + c2;
std::cout << c3.real << " + " << c3.imag << "i" << std::endl;
```

```
8 + 10i
```

Control Statements

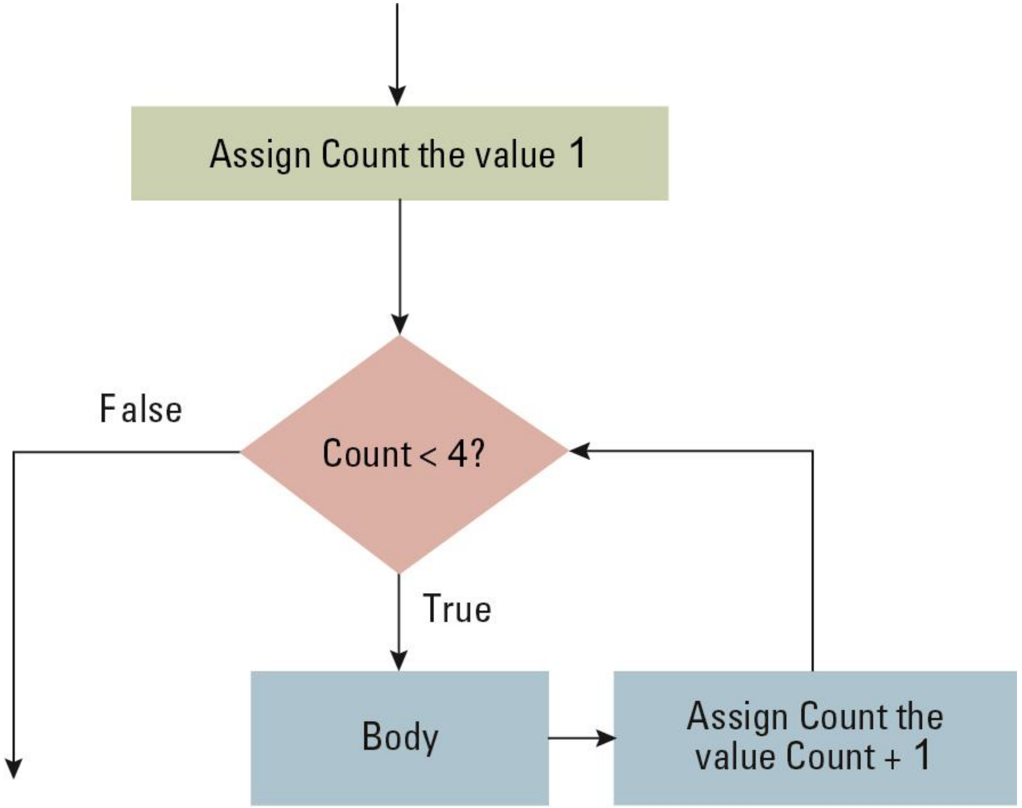
- **Old-fashion: goto**
  - Not recommended

```
line #   goto 4
  2      print "passed."
         goto 7
  4      if (grade < 60) goto 6
         goto 2
  6      print "failed."
  7      stop
```

- Modern programming
  - if / else if / else
  - switch
  - for
  - while

Control Statements (cont.)

- for




```
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

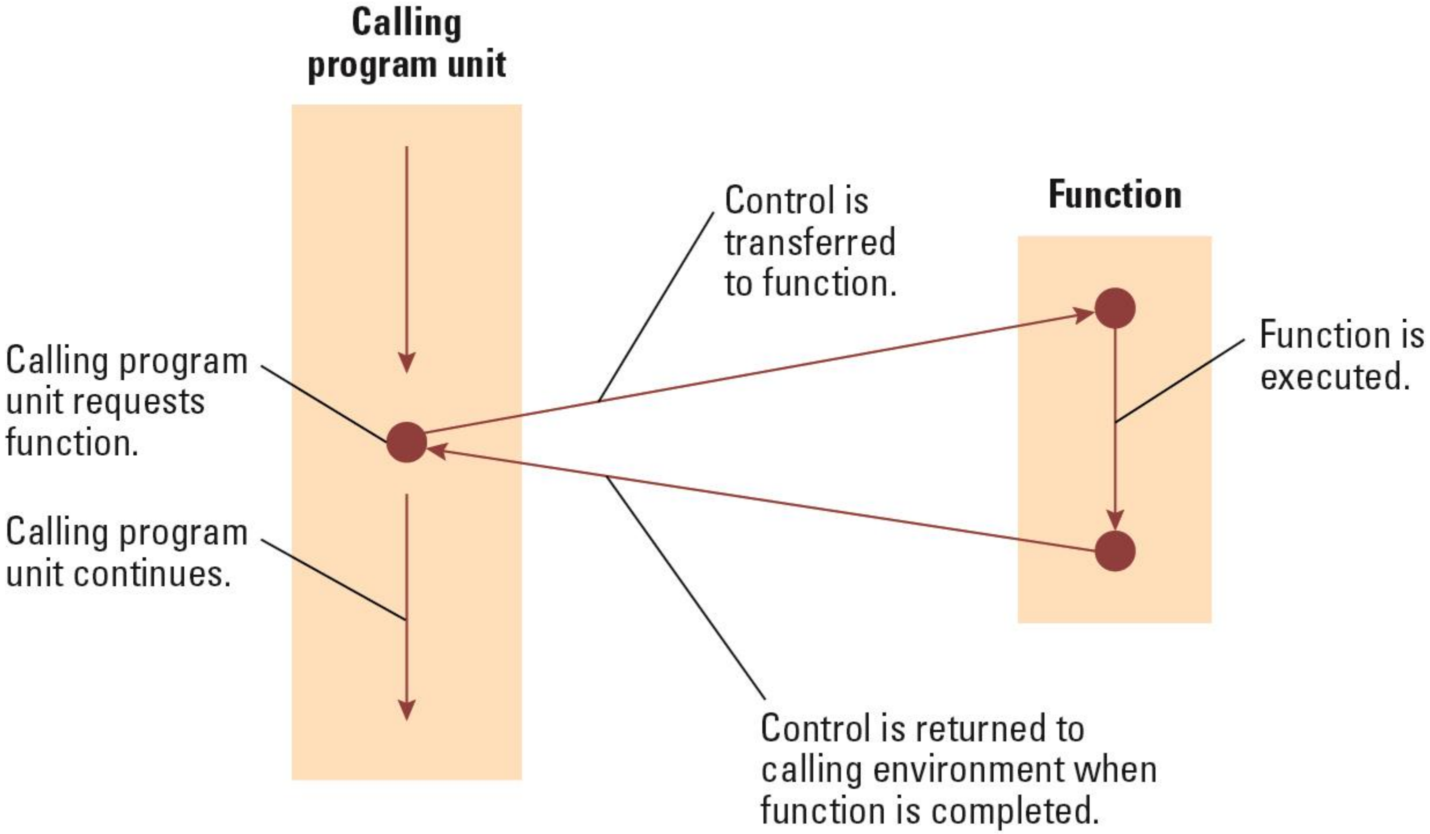
```
a = b + c;  // End-of-line comment.

/* Block comment */
a = b + c;

/**
    Documentation comment.
*/
a = b + c;
```

Outline

- Historical perspective
- Traditional programming concepts
- Procedural units
- Language translation process
- Object-oriented programming
- Programming concurrent activities

Procedural Units

- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function



Procedural Units (cont.)

- Terminology

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.

type of the return value

formal parameters

function header

local variable

```
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.

Procedural Units (cont.)

- Terminology

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

type of the return value

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

{float Volume;

 Volume = 3.14 * Radius * Radius * Height;

 return Volume;

}
```

Declare a local variable named Volume.

Compute the volume of the cylinder.

return value

Terminate the function and return the value of the variable Volume.

Procedural Units (cont.)

- Function's (procedure's) header

```
void Swap(int*, int*);
```

can be put in another header file

```
int a = 5;
int b = 3;
Swap(&a, &b);
std::cout << a << " " << b << std::endl;

void Swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Procedural Units (cont.)

- **Local variable** and **global variables**

```
// Global variable.
int var = 0;

int main()
{
    // Local variable.
    int var = 5;

    std::cout << var << std::endl;      → 5
    std::cout << ::var << std::endl;    → 0
```

Procedural Units (cont.)

- **Formal parameters** and **actual parameters**

```
void Swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int x = 5;
    int y = 3;
    Swap(&x, &y);
    std::cout << x << " " << y << std::endl;
```

a, b: formal parameters

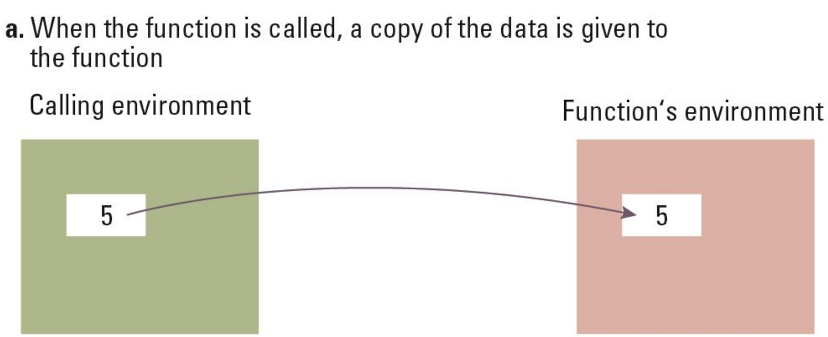
x, y: actual parameters

Procedural Units (cont.)

- **Passing parameters**
  - Call by **value** (passed by value)
  - Call by **reference** (passed by reference)
  - Call by **address** (a variant of call-by-reference)

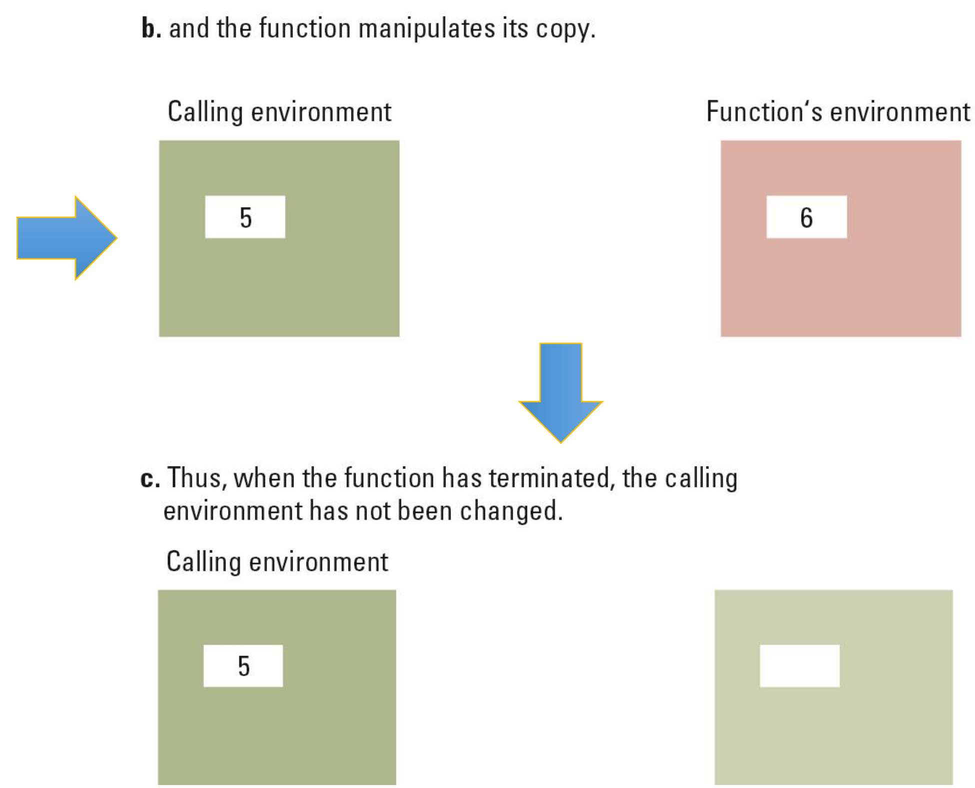
Procedural Units (cont.)

- **Passing parameters**
  - Call by **value** (passed by value)




```
void Test(int v)
{
    v = 6;
}

int main()
{
    int val = 5;
    Test(val);
    std::cout << val << std::endl;
```

5



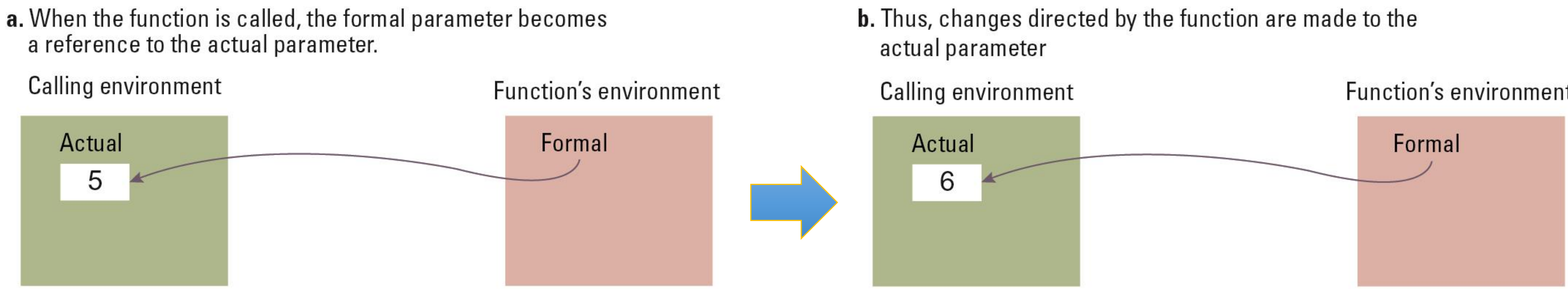
Procedural Units (cont.)

- **Passing parameters**
  - Call by **address** (passed by address)

```
void Test(int *v)
{
    *v = 6;
}

int main()
{
    int val = 5;
    Test(&val);
    std::cout << val << std::endl;   6
```

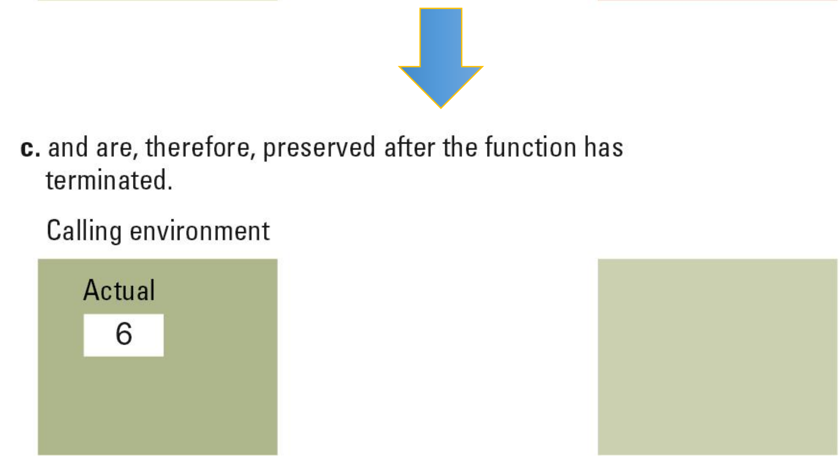
Procedural Units (cont.)

- **Passing parameters**
  - Call by **reference** (passed by reference)




```
void Test(int &ref_v)
{
    ref_v = 6;
}

int main()
{
    int val = 5;
    Test(val);
    std::cout << val << std::endl;
```

6

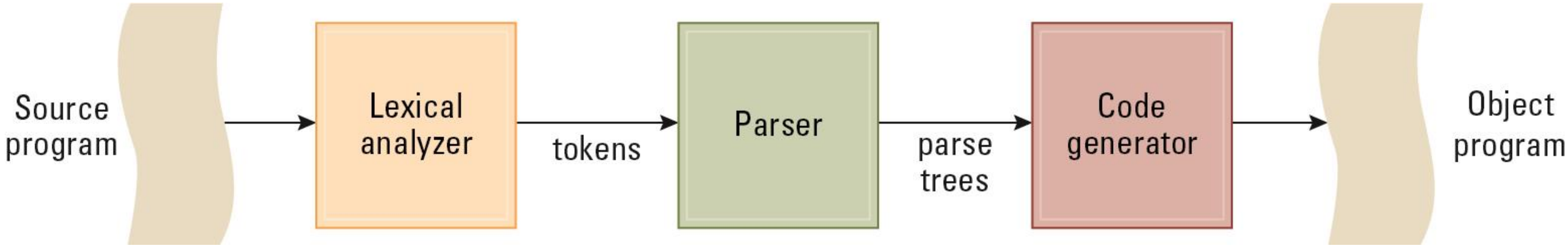


Outline

- Historical perspective
- Traditional programming concepts
- Procedural units
- Language translation process
- Object-oriented programming
- Programming concurrent activities

Language Translation Process

- Converting a program written in a high-level language into a machine-executable form



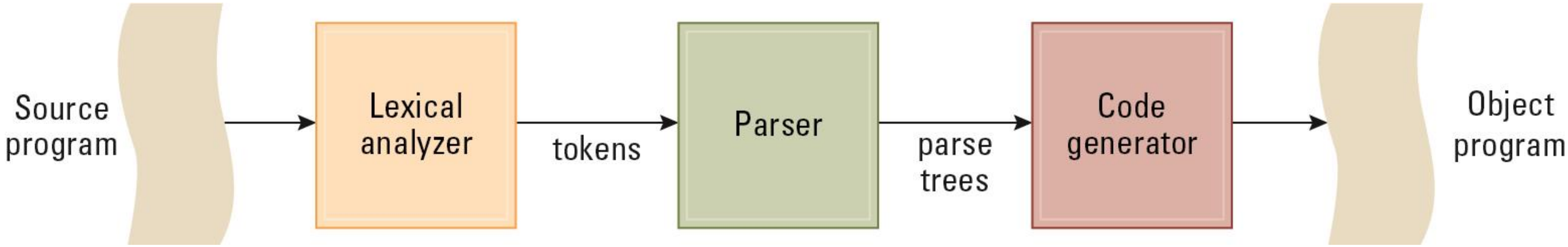
- **Lexical Analyzer:** recognize which strings of symbols represent a single entity, or token (identify tokens)

$\boxed{y}\ \boxed{=}\ \boxed{x}\ \boxed{+}\ \boxed{1000}$

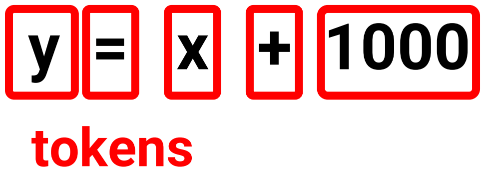
tokens

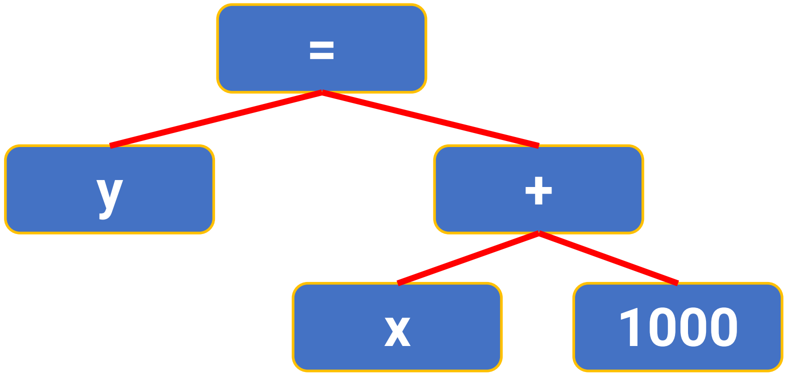
Language Translation Process (cont.)

- Converting a program written in a high-level language into a machine-executable form



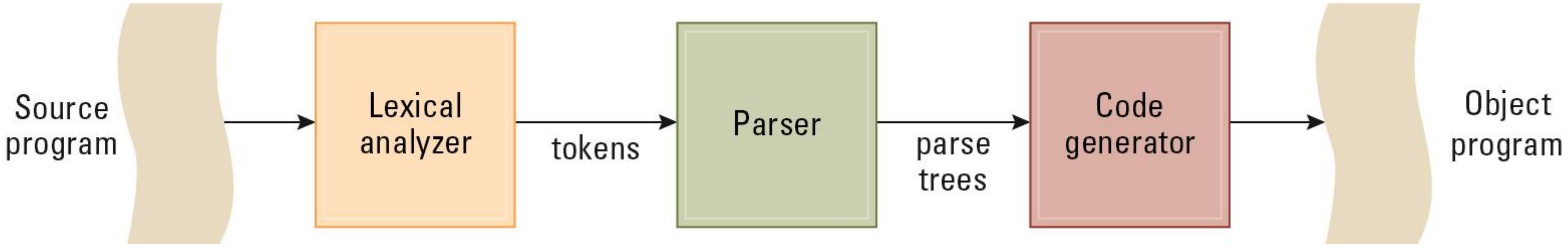
  - **Parser:** group tokens into statements, using syntax diagrams to make parse trees (identify syntax)





Language Translation Process (cont.)

- Converting a program written in a high-level language into a machine-executable form



  - **Code Generator:** construct machine-language instructions to implement the statements
    - Link libraries

Syntax Grammar for Algebra

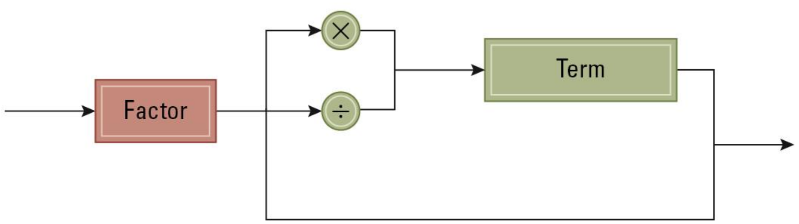
- A simple syntax grammar:

Expression → Term | Term + Expression
| Term – Expression

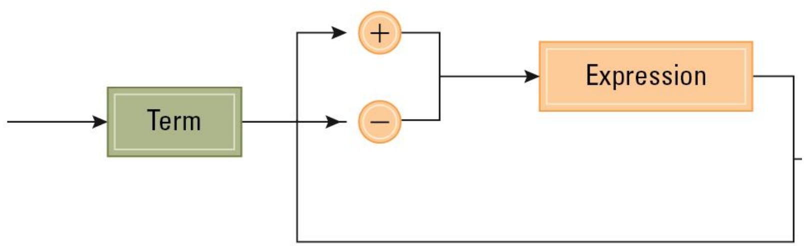
Term → Factor | Factor * Term | Factor / Term

Factor → x | y | z

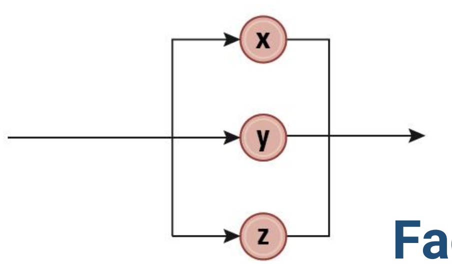
expression, term, factor: terminals
x, y, z, +, -, *, / : non-terminals



Term



Expression



Factor

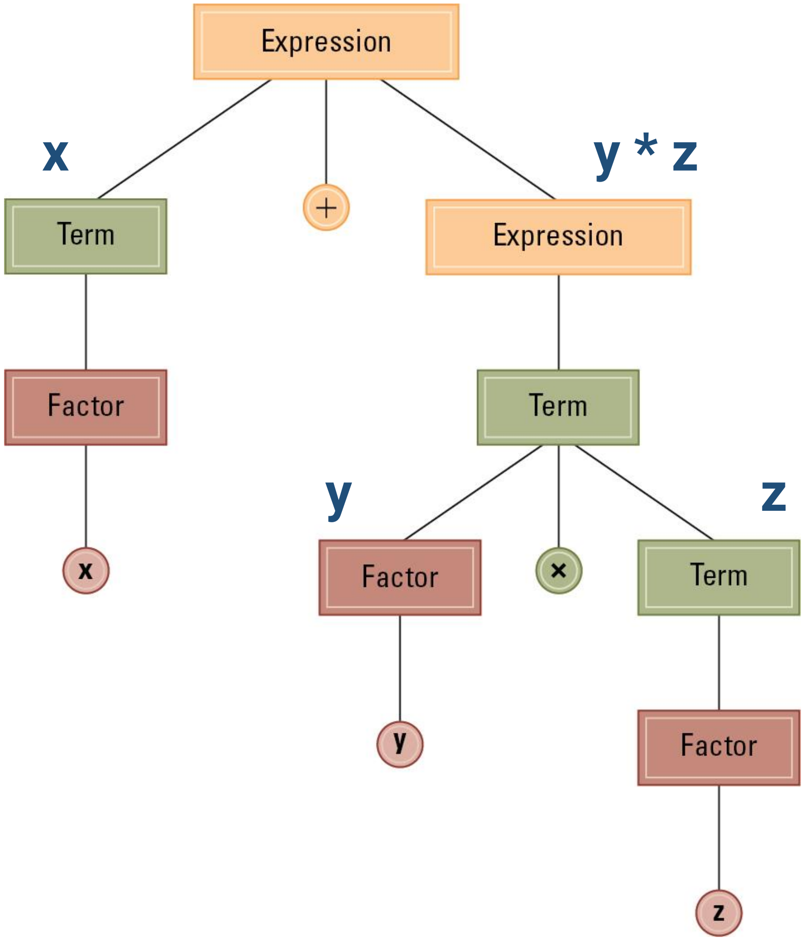
Syntax Grammar for Algebra (cont.)

- Example: is **x + y * z** an expression?

Expression → Term |
Term + Expression |
Term – Expression

Term → Factor |
Factor * Term |
Factor / Term

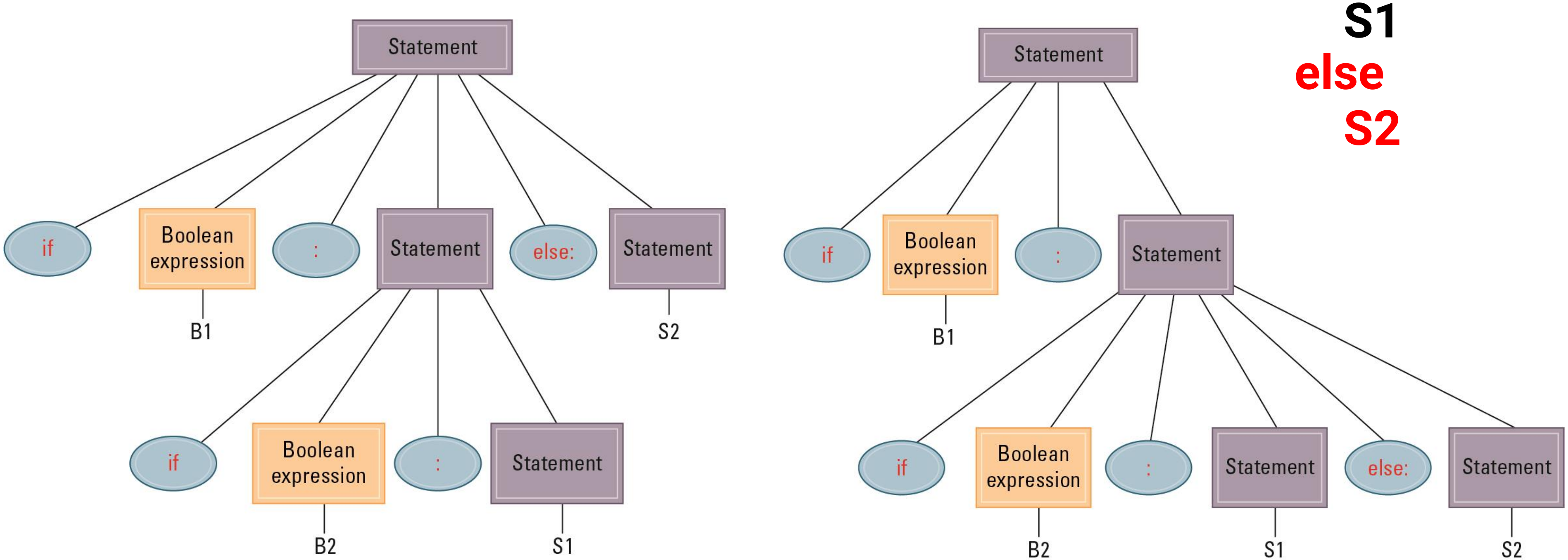
Factor → x |
y |
z



Ambiguity

- **if** B1 **then** **if** B2 **then** S1 **else** S2

if (B1)
if (B2)
S1
else
S2



Code Generation

- **Coercion:** implicit conversion between data types
- **Strongly typed**
  - No coercion, data types must agree with each other
  - Handle type conversion by programmers
- **Code optimization**

```
x = y + z;
w = x + z;
```

```
w = y + (z << 1);
```

Outline

- Historical perspective
- Traditional programming concepts
- Procedural units
- Language translation process
- Object-oriented programming
- Programming concurrent activities

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.

Components of an Object

- **Instance variable (member variable)**
  - Variable within an object
  - Holds information within the object
- **Method (member function)**
  - 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
- **Destructor** v.s. **garbage collection**

Components of an Object (cont.)

- An example of Class

```
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.

Object Integrity

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

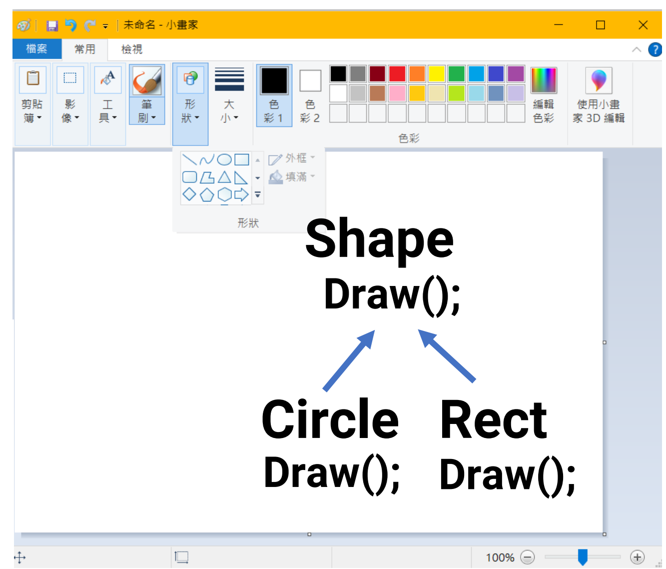
Components in the class are designated public or private depending on whether they should be accessible from other program units.

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

Additional Object-oriented Concepts

- **Inheritance**
  - Allows new classes to be defined in terms of previously defined classes

```
class Shape {
public:
    Shape(){}
    ~Shape(){}
    virtual void Draw() = 0;
};
class Circle : public Shape {
public:
    Circle(){}
    ~Circle(){}
    void Draw() { std::cout << "Draw Circle!" << std::endl; }
};
class Rect : public Shape {
public:
    Rect(){}
    ~Rect(){}
    void Draw() { std::cout << "Draw Rect!" << std::endl; }
};
```

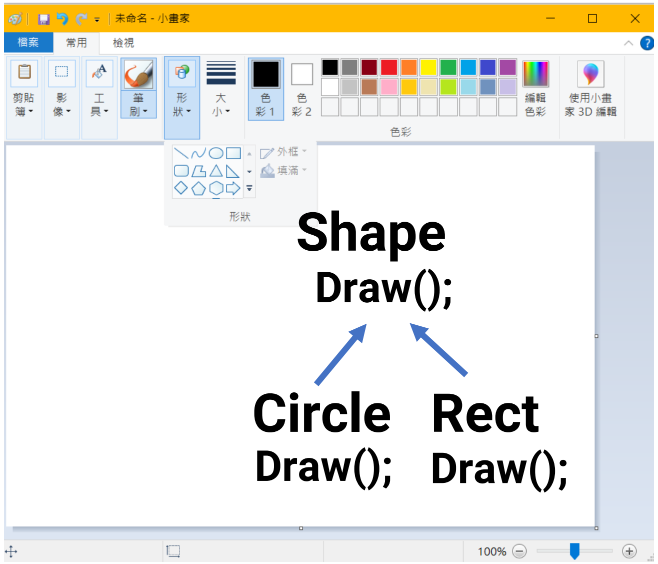


Additional Object-oriented Concepts

- **Polymorphism**
  - Allows method calls to be interpreted by the object that receives the call

```cpp
Shape* shapeList[2];
shapeList[0] = new Circle();
shapeList[1] = new Rect();
for (int i = 0; i < 2; ++i) {
    shapeList[i]->Draw();
}
```

```
Draw Circle!
Draw Rect!
```

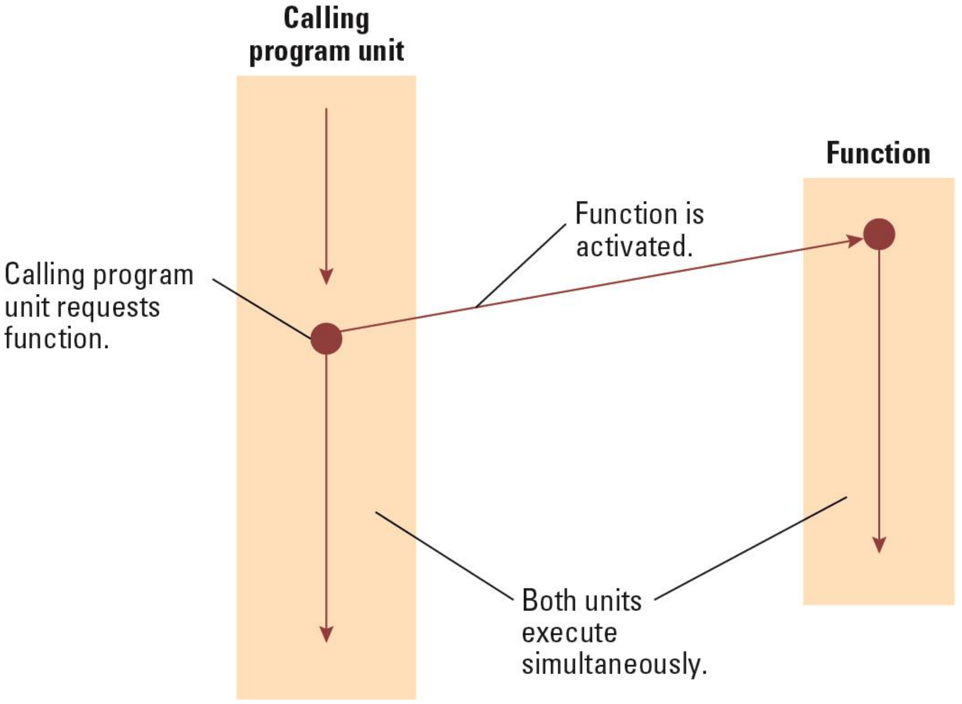


Outline

- Historical perspective
- Traditional programming concepts
- Procedural units
- Language translation process
- Object-oriented programming
- Programming concurrent activities

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



Any Questions?