Universidad Nacional de Ingeniería
Facultad de Ciencias

Introducción a la Ciencia de la Computación

Lenguajes de Programacion

Prof: J. Solano
2011-I

Objetivos

Después de estudiar este capítulo el estudiante sera capaz de:

- **Describir la evolución de los lenguajes de programación de lenguaje de máquina a lenguajes de alto nivel.**
- **Entender cómo un programa en un lenguaje de alto nivel se traduce en lenguaje de máquina.**
- **Distinguir entre los cuatro paradigmas de lenguajes de programación.**
- **Entender el paradigma de procedimientos y la interacción entre una unidad de programa y los datos de los elementos en el paradigma.**
- **Entender el paradigma orientado a objetos y la interacción entre una unidad de programa y los objetos en este paradigma.**
- **Definir paradigma funcional y comprender sus aplicaciones.**
- **Definir un paradigma de la declaración y comprender sus aplicaciones.**
- **Definir conceptos comunes en lenguajes de procedimiento y orientado a objetos.**

EVOLUCIÓN

Para escribir un programa para un ordenador, debemos utilizar un lenguaje de programación. Un lenguaje de programación es un conjunto de palabras predefinidas que se combinan en un programa de acuerdo a reglas predefinidas (***syntax***). Con los años, los lenguajes de programación han evolucionado a partir de ***lenguaje de máquina*** a ***lenguajes de alto nivel***.

Lenguajes de máquina

En los primeros días de las computadoras, los unicos lenguajes de programación disponibles eran **lenguajes de máquina**. Cada computador tenia su propio lenguaje de máquina, hecho de patrones (streams) de 0s y 1s. En el capítulo anterior se demostró que en una computadora hipotética primitiva, tenemos que utilizar once líneas de código para leer dos números enteros, añadirlos e imprimir el resultado. Estas líneas de código, cuando estás escritas en lenguaje de máquina, hacen once líneas de código binario, cada una de 16 bits.

El único lenguaje que entiende un computador es el lenguaje de máquina.

Code in machine language to add two integers

Hexadecimal	*Code in machine language*			
$(1FEF)_{16}$	0001	1111	1110	1111
$(240F)_{16}$	0010	0100	0000	1111
$(1FEF)_{16}$	0001	1111	1110	1111
$(241F)_{16}$	0010	0100	0001	1111
$(1040)_{16}$	0001	0000	0100	0000
$(1141)_{16}$	0001	0001	0100	0001
$(3201)_{16}$	0011	0010	0000	0001
$(2422)_{16}$	0010	0100	0010	0010
$(1F42)_{16}$	0001	1111	0100	0010
$(2FFF)_{16}$	0010	1111	1111	1111
$(0000)_{16}$	0000	0000	0000	0000

Lenguajes Ensambladores

La siguiente evolución en programación vino con la idea de reemplazar el código binario por instrucciones y direcciones con símbolos o mnemónicos. Debido a que utilizaban símbolos, estos lenguajes fueron primeramente conocidos como lenguajes simbólicos. El conjunto de estos lenguajes mnemónicos fueron más tarde conocidos como lenguajes ensambladores. El lenguaje ensamblador para nuestro ordenador hipotético que reemplaza el lenguaje de máquina se muestra en la siguiente tabla.

El único lenguaje que entiende un computador es el lenguaje de máquina.

Code in assembly language to add two integers

Code in assembly language				*Description*
LOAD	**RF**	**Keyboard**		Load from keyboard controller to register F
STORE	**Number1**	**RF**		Store register F into Number1
LOAD	**RF**	**Keyboard**		Load from keyboard controller to register F
STORE	**Number2**	**RF**		Store register F into Number2
LOAD	**R0**	**Number1**		Load Number1 into register 0
LOAD	**R1**	**Number2**		Load Number2 into register 1
ADDI	**R2**	**R0**	**R1**	Add registers 0 and 1 with result in register 2
STORE	**Result**	**R2**		Store register 2 into Result
LOAD	**RF**	**Result**		Load Result into register F
STORE	**Monitor**	**RF**		Store register F into monitor controller
HALT				Stop

Lenguajes de alto nivel

Aunque los lenguajes ensambladores mejorado mucho la eficiencia de programación, todavía requeria que los programadores se concentrasen en el hardware que utilizaban. Trabajar con lenguajes simbólicos también era muy tedioso, ya que cada instrucción de máquina tenía que ser codificada individualmente. El deseo de mejorar la eficiencia del programador y cambiar el enfoque desde el ordenador al problema a resolver dio lugar al desarrollo de lenguajes de alto nivel.

Con los años, varios lenguajes, sobre todo BASIC, COBOL, Pascal, Ada, C, C++ y Java, fueron desarrollados. El siguiente programa muestra el código para sumar dos números enteros tal y como aparecería en el lenguaje C++.

Addition program in C++

```
/*      This program reads two integers from keyboard and prints their sum.
        Written by:
        Date:
*/
#include <iostream.h>
using namespace std;
int main (void)
{
        // Local Declarations
        int number1;
        int number2;
        int result;
        // Statements
        cin >> number1;
        cin >> number2;
        result = number1 + number2;
        cout << result;
        return 0;
} // main
```

TRADUCCIÓN

Los programas de hoy en día se escriben normalmente en uno de los lenguajes de alto nivel. Para ejecutar el programa en un ordenador, el programa debe ser traducido al lenguaje de máquina del equipo en el que se ejecutará. El programa en un lenguaje de alto nivel se llama el programa fuente. El programa traducido en lenguaje de máquina que se llama el programa objeto. Se utilizan dos métodos para la traducción: **compilación** e **interpretación**.

Compilación

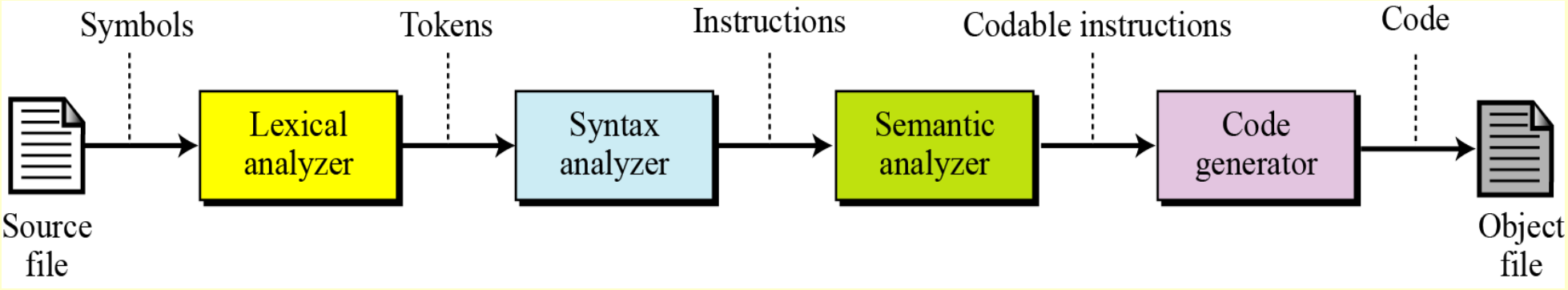
Un compilador traduce normalmente el **código fuente** completo en el **programa objeto**.

Interpretación

Algunos lenguajes de programación usan un intérprete para traducir el código fuente fuente en el programa objeto. La interpretación se refiere al proceso de traducir cada línea del código fuente en la línea correspondiente del programa objeto y la ejecución de la línea. Sin embargo, tenemos que ser conscientes de dos tendencias en la interpretación: el que utilizan algunos lenguajes antes de Java y la interpretación utilizada por Java.

Proceso de traducción

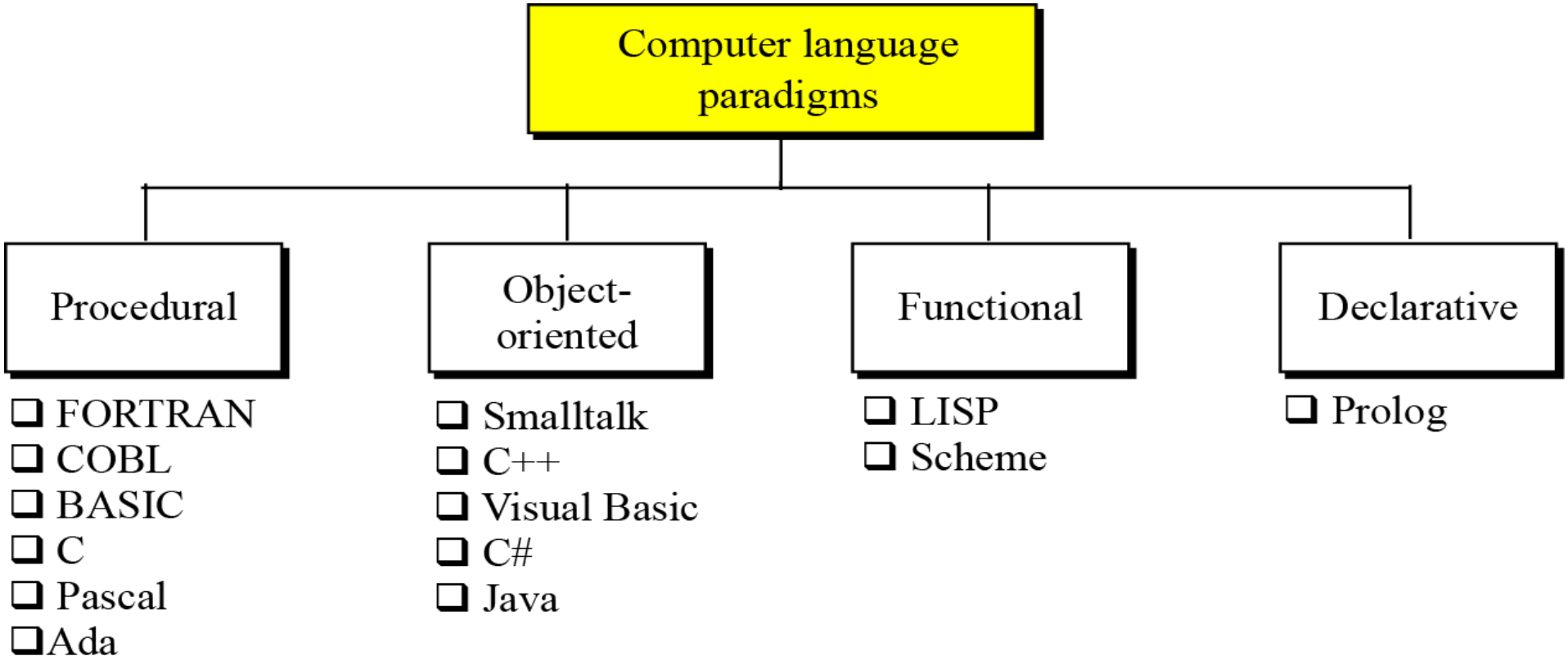
Compilación e interpretación se diferencian en que el primero traduce el código fuente completo antes de ejecutarlo, mientras que el segundo se traduce y ejecuta el código fuente de una línea a la vez. Ambos métodos, sin embargo, siguen el mismo proceso de traducción mostrado en la figura.



Proceso de traducción del código fuente

PARADIGMAS DE PROGRAMACIÓN

Hoy en día, los lenguajes de programación se clasifican de acuerdo con el enfoque que utilizan para resolver un problema. Un **paradigma**, por lo tanto, es una forma en la que un lenguaje de computador analiza el problema a resolver. Dividimos los lenguajes de programación en cuatro paradigmas: de ***procedimiento***, ***orientado a objetos***, ***funcional*** y ***declarativo***.

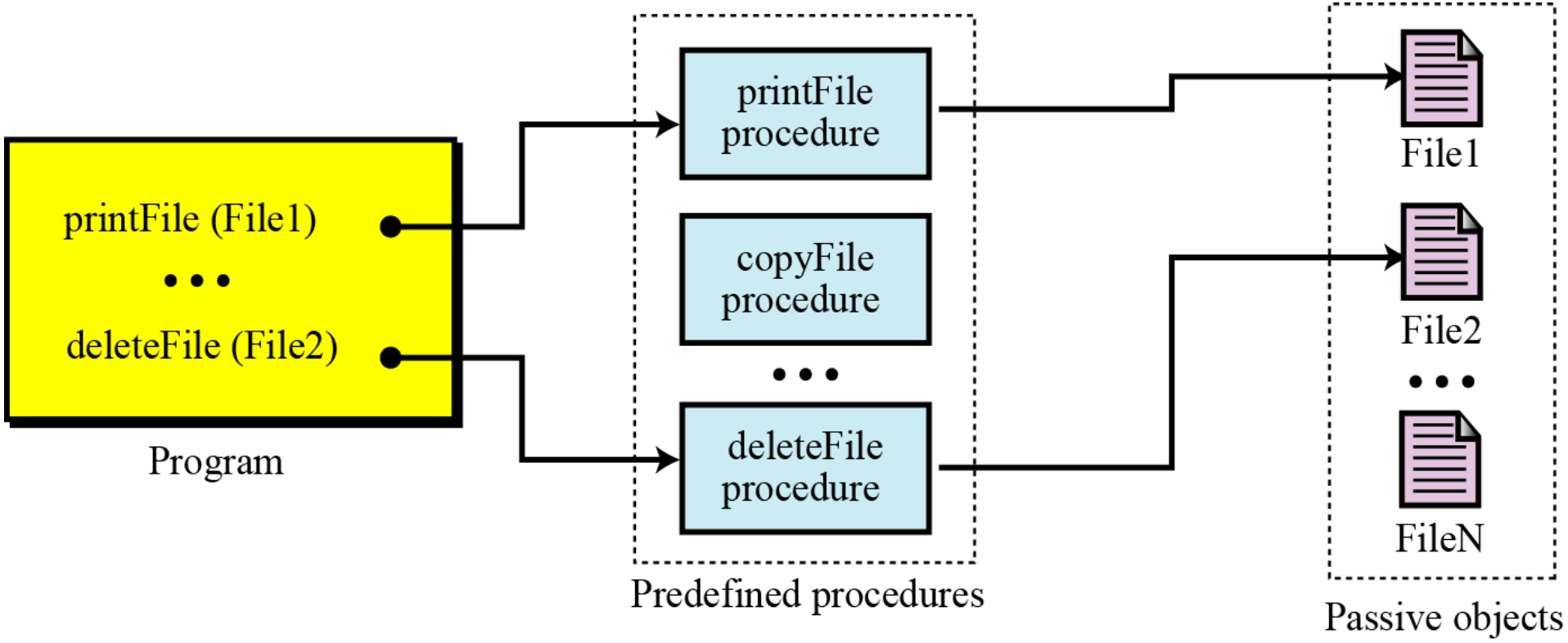


Categorías de lenguajes de programación

El paradigma de procedimientos

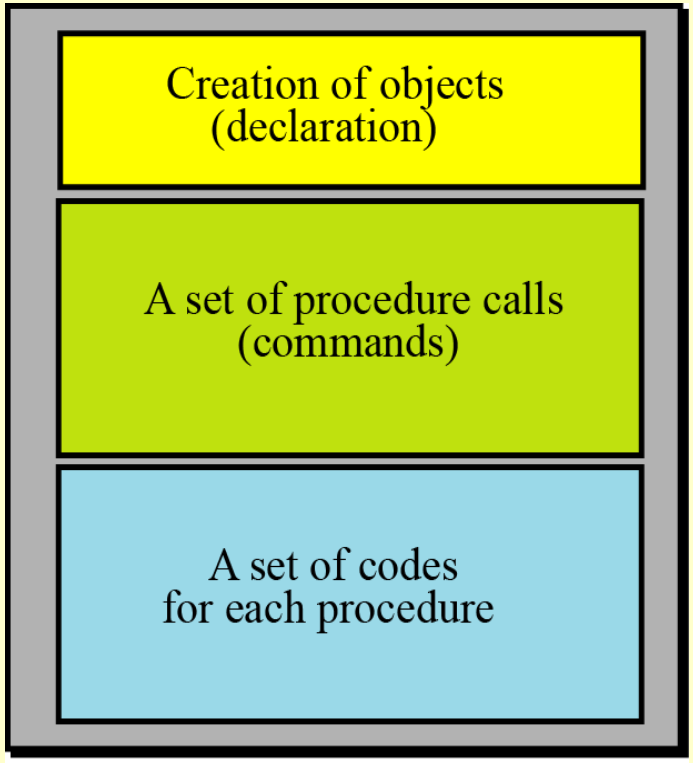
En el paradigma de procedimiento (o paradigma imperativo) podemos pensar en un programa como un agente activo que manipula objetos pasivos. Nos encontramos con muchos objetos pasivos en nuestra vida diaria: una piedra, un libro, una lámpara, y así sucesivamente. Un objeto pasivo no puede iniciar una acción por sí misma, pero puede recibir acciones de los agentes activos.

Un programa en un paradigma de procedimiento es un agente activo que utiliza objetos pasivos que nos referimos como datos o elementos de datos. Para manipular una pieza de datos/información, el agente activo (programa) emite una acción, referida como un **procedimiento**. Por ejemplo, piense en un programa que imprime el contenido de un archivo. El archivo es un objeto pasivo. Para imprimir el archivo, el programa utiliza un procedimiento, que llamamos de impresión.



El concepto del paradigma de procedimiento

Un programa en este paradigma se compone de tres partes: *una parte para la creación de objetos*, *un conjunto de llamadas de procedimiento* y *un conjunto de código para cada procedimiento*. Algunos de los procedimientos ya han sido definidas en el propio idioma. Mediante la combinación de este código, el programador puede crear nuevos procedimientos.



Componentes de un programa de procedimientos

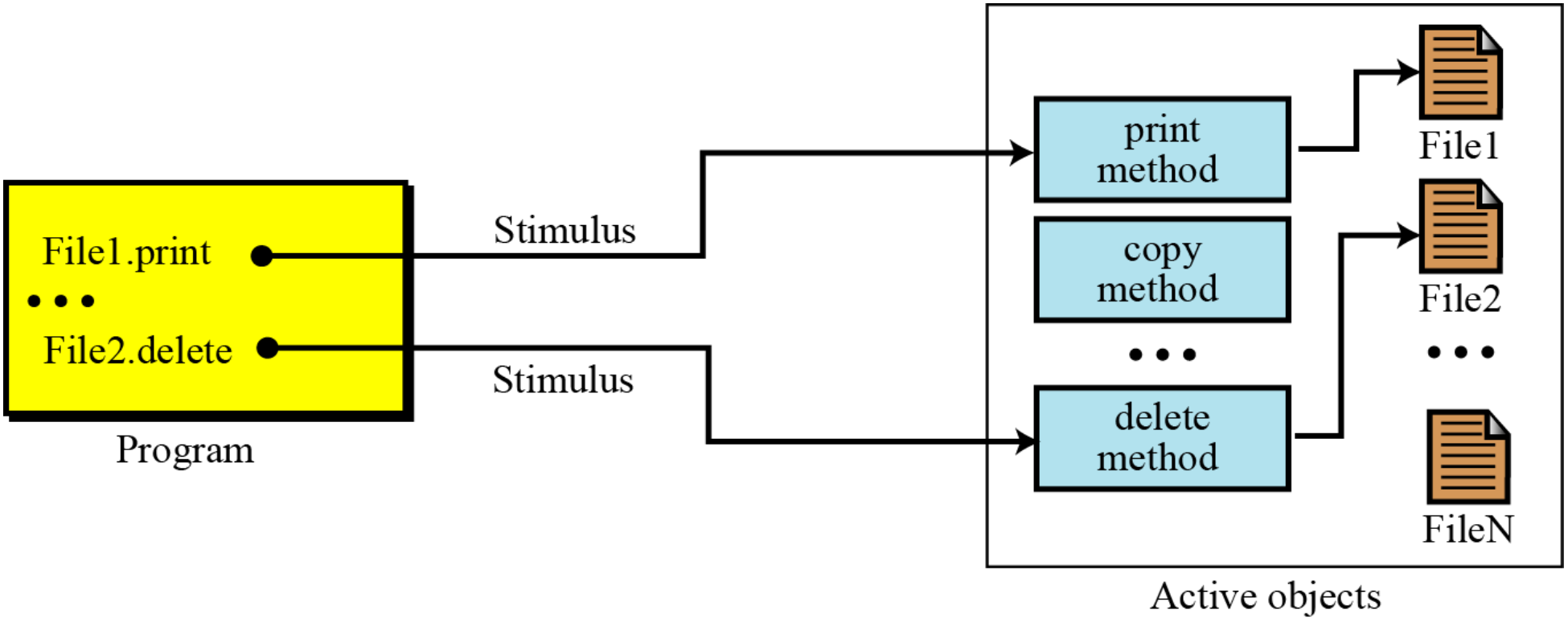
Algunos lenguajes de procedimientos

- FORTRAN (FORmula TRANslation)
- COBOL (Common Business-Oriented Languages)
- Pascal
- C
- Ada

El paradigma orientado a objetos

El paradigma orientado a objetos trabaja con objetos activos en lugar de objetos pasivos. Nos encontramos con muchos objetos activos en nuestra vida cotidiana: un vehículo, una puerta automática, lavavajillas, etc. La acción a realizarse sobre estos objetos se incluyen en el objeto: los objetos sólo tienen que recibir el estímulo apropiado desde el exterior para llevar a cabo una de las acciones.

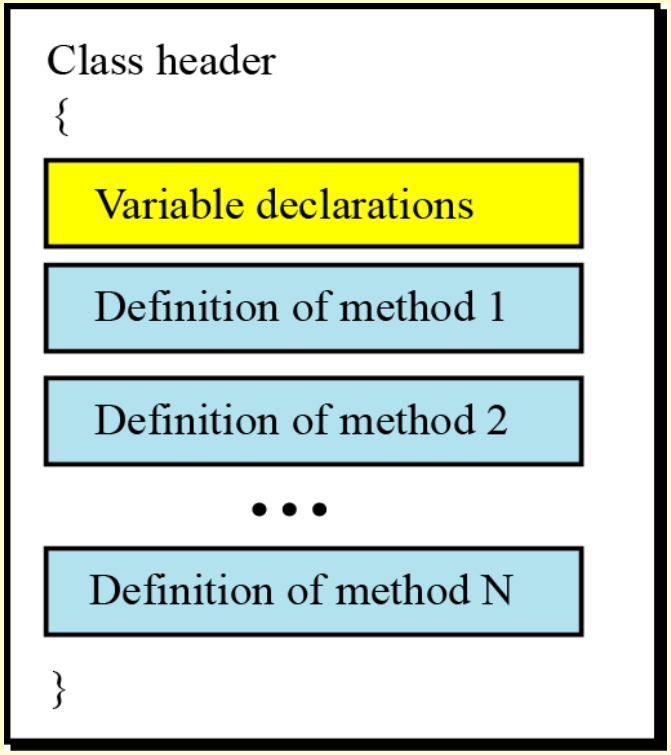
Un archivo en un paradigma orientado a objetos se puede empaquetar con todos los procedimientos--llamados métodos en el paradigma orientado a objetos—a ser llevados a cabo por el archivo: impresión, copia, eliminación y así sucesivamente. El programa de este paradigma sólo envía la solicitud correspondiente al objeto.



El concepto del paradigma orientado a objetos

Clases

Como muestra la figura, objetos del mismo tipo (archivos, por ejemplo) necesitan un conjunto de métodos que muestran cómo un objeto de este tipo reacciona a los estímulos del exterior de los "territorios" del objeto. Para crear estos métodos, se utiliza una unidad llamada clase.



Concepto de un programa orientado a objetos

Métodos

En general, el formato de los métodos son muy similares a las funciones utilizadas en algunos lenguajes de procedimiento. Cada método tiene su cabecera, sus variables locales y su declaración. Esto significa que la mayoría de las características que discutimos para lenguajes de procedimiento se aplican también a los métodos escritos para un programa orientado a objetos. En otras palabras, podemos afirmar que lenguajes orientados a objetos son en realidad una extensión de las lenguas de procedimiento con algunas nuevas ideas y nuevas características. El lenguaje C + +, por ejemplo, es una extensión orientada a objetos del lenguaje C.

Herencia

En el paradigma orientado a objetos, como en la naturaleza, un objeto puede heredar de otro objeto. Este concepto se denomina herencia. Cuando una clase general se define, podemos definir una clase más específica, que hereda algunas de las características de la clase general, pero también tiene algunas características nuevas. Por ejemplo, cuando un objeto del tipo GeometricalShapes es definido, podemos definir una clase llamada Rectangles. Los rectángulos son figuras geométricas con características adicionales.

Polimorfismo

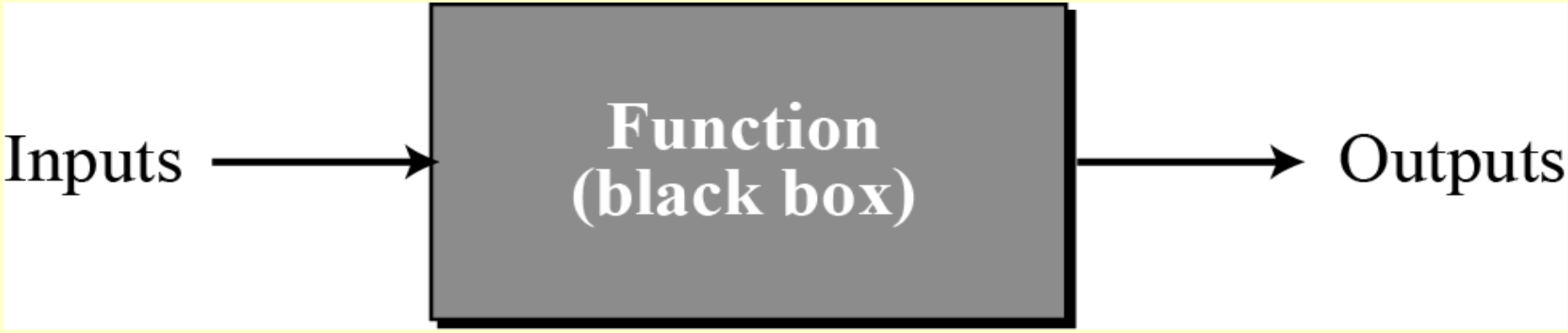
Polimorfismo significa "muchas formas". Polimorfismo en el paradigma orientado a objetos significa que podemos definir varias operaciones con el mismo nombre que puede hacer cosas diferentes en clases relacionadas. Por ejemplo, supongamos que se definen dos clases, Rectangles y Circles, ambos heredados de la clase GeometricalShapes. Definimos dos operaciones, ambas llamadas area, una en Rectangles y una en Circles, que calculan el área de un rectángulo o un círculo. Las dos operaciones tienen el mismo nombre.

Algunos lenguajes orientados a objetos

- C++
- Java

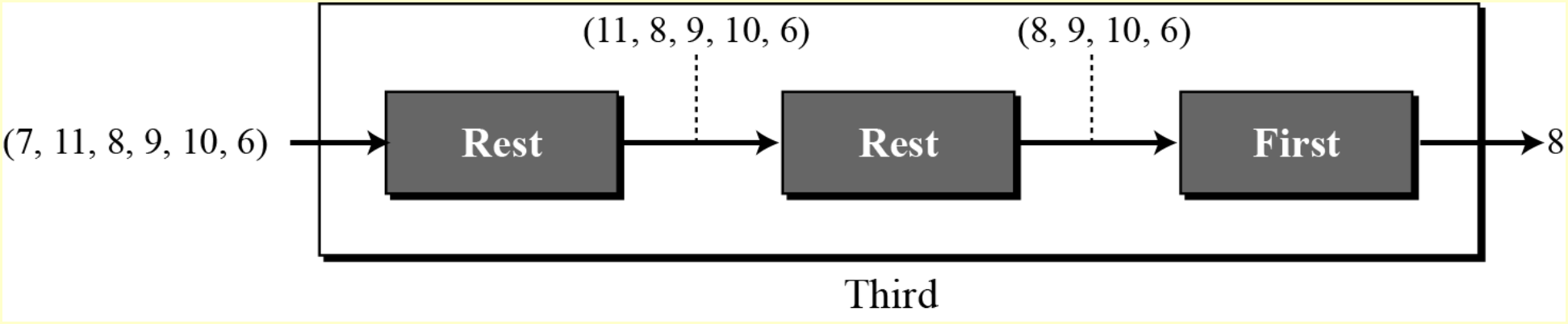
El paradigma funcional

En el paradigma funcional un programa es considerado una función matemática. En este contexto, una función es una caja negra que asigna (mapea) una lista de entrada de una lista de salida/resultados.



Una función en un lenguaje funcional

Por ejemplo, podemos definir una funcion primitiva llamada First que extrae el primer elemento de una lista. Tambien puede haber una funcion llamada Rest que extrae todos los elementos excepto el primero. Un programa puede definir una funcion que extrae el tercer elemento de una lista combinando las otras dos funciones, como se muestra en la figura.



Extrayendo el tercer elemento de una lista

Algunos lenguajes funcionales

- LISP (LISt Programming)
- Scheme

El paradigma declarativo

Un paradigma declarativo utiliza el principio de razonamiento lógico para responder a las consultas. Se basa en la lógica formal definida por los matemáticos griegos y que más tarde se convirtió en el cálculo de predicados de primer orden.

El razonamiento lógico se basa en la deducción. Algunas de las afirmaciones (hechos) se asume que son verdad, y el lógico hace uso de normas sólidas de razonamiento lógico para deducir nuevas declaraciones (hechos). Por ejemplo, la famosa regla de deducción en lógica es la siguiente:

If (A is B) and (B is C), then (A is C)

Usando esta regla y las dos afirmaciones siguientes,

Fact 1: Socrates is a human → A is B
Fact 2: A human is mortal → B is C

podemos deducir una nueva afirmación:

Fact 3: Socrates is mortal → A is C

Prolog

Uno de los lenguajes declarativos mas famosos es Prolog (PROgramming in LOGic), desarrollado por A. Colmerauer en Francia en 1972. Un programa en Prolog se compone de hechos y reglas. Por ejemplo, las afirmaciones previas sobre los seres humanos se pueden establecer como:

```
human (John)
mortal (human)
```

Entonces el usuario puede preguntar:

```
?-mortal (John)
```

y el programa va a responder SI.

CONCEPTOS COMUNES

En esta sección realizamos una navegación rápida a través de algunos lenguajes de procedimiento para encontrar conceptos comunes. Algunos de estos conceptos también están disponibles en la mayoría de los lenguajes orientados a objetos, ya que, como hemos explicado, el paradigma orientado a objetos utiliza el paradigma de procedimiento para la creación de métodos.

Identificadores

Una característica presente en todos las lenguajes de procedimiento, así como en otros lenguajes, es el **identificador**, es decir, el nombre de los objetos. Identificadores nos permiten nombrar objetos en el programa. Por ejemplo, cada pieza de datos en un ordenador se almacena en una dirección única. Si no hubiera identificadores para representar ubicaciones de datos simbólicamente, tendríamos que conocer y utilizar las direcciones de datos para manipularlos. En cambio, simplemente damos nombres de datos y dejamos al compilador hacer un seguimiento de dónde están esos datos ubicados físicamente.

Tipos de datos

Un **tipo de datos** define un conjunto de valores y un conjunto de operaciones que se pueden aplicar a esos valores. El conjunto de valores para cada tipo se conoce como el dominio para el tipo. La mayoría de lenguajes definen dos categorías de tipos de datos: los *tipos simples* y *tipos compuestos*.

Un tipo simple es un tipo de datos que no se puede dividir en tipos de datos más pequeños.

Un tipo compuesto es un conjunto de elementos en los que cada elemento es un tipo simple o un tipo de compuesto.

Tipos de datos

Variables son nombres de lugares de memoria. Como se discutió en el capítulo anterior, cada posición de memoria en un ordenador tiene una dirección. Aunque las direcciones son utilizadas por el computador internamente, es muy incómodo para el programador usar las direcciones. Un programador puede utilizar una variable, como la nota (score), para almacenar el valor entero de una nota recibida en un examen. Dado que una variable contiene un elemento de datos, entonces tiene un tipo.

Literales

Un literal es un valor predeterminado utilizado en un programa. Por ejemplo, si tenemos que calcular el área de un círculo cuando el valor del radio se almacena en la variable r, podemos utilizar la expresión $3.14 \times r^2$, en el que se utiliza el valor aproximado de π (pi) como un literal. En la mayoria de lenguajes de programación podemos tener entero, real, carácter y literales booleanos. En la mayoría de lenguajes, también podemos tener cadenas de literales. Para distinguir los caracteres y cadenas literales de los nombres de variables y otros objetos, la mayoría de los lenguajes requieren que los caracteres literales esten entre comillas simples, como 'A', y las cadenas encerradas entre comillas dobles, como "Anne".

Constantes

El uso de literales no se considera buena práctica de programación, a menos que esté seguro de que el valor del literal no va a cambiar con el tiempo (por ejemplo, el valor de π en la geometría). Sin embargo, la mayoría de los literales puede cambiar el valor con el tiempo.

Por esta razón, la mayoría de los lenguajes de programación definir constantes. Una **constante**, como una variable, es un lugar con nombre que puede almacenar un valor, pero el valor no se puede cambiar después de que se ha definido al principio del programa. Sin embargo, si queremos usar el programa más tarde, podemos cambiar una sola línea al comienzo del programa, el valor de la constante.

Inputs y Outputs

Casi todos los programas tiene que leer y/o escribir datos. Estas operaciones pueden ser bastante complejas, sobre todo cuando leemos y escribimos archivos de gran tamaño. La mayoría de los lenguajes de programación utilizan una función predefinida para entrada y salida.

Los datos son ingresados (input) por una instrucción o una función predefinida como ***scanf*** en lenguaje C.

Los datos son egresados (output) por una instrucción o una función predefinida como ***printf*** en lenguaje C.

Expresiones

Una expresión es una secuencia de operandos y operadores que se reduce a un solo valor. Por ejemplo, la siguiente es una expresión con un valor de 13:

```
2 * 5 + 3
```

Un operador es un símbolo (token) de un lenguaje específico que requiere una acción a tomar. Los operadores más conocidos provienen de la Matemática.

Esta tabla muestra algunas operaciones aritmeticas usadas en C, C++, y Java

Arithmetic operators

Operator	*Definition*	*Example*
+ – * / %	Addition Subtraction Multiplication Division (the result is the quotient) Division (the result is the remainder)	3 + 5 2 – 4 Num * 5 Sum / Count Count % 4
++ – –	Increment (add 1 to the value of the variable) Decrement (subtract 1 from the value of the variable)	Count++ Count– –

Operadores relacionales comparan datos para ver si un valor es mayor, menor o igual a/que otro valor. El resultado de aplicar los operadores relacionales es un valor booleano/lógico (verdadero o falso). C, C++ y Java utilizan seis operadores de relación, como se muestra en esta tabla

Relational operators

Operator	*Definition*	*Example*
<	Less than	Num1 < 5
<=	Less than or equal to	Num1 <= 5
>	Greater than	Num2 > 3
>=	Greater than or equal to	Num2 >= 3
==	Equal to	Num1 == Num2
!=	Not equal to	Num1 != Num2

Operadores lógicos combinan valores booleanos (verdadero o falso) para obtener un nuevo valor. El lenguaje C utiliza tres operadores lógicos, como se muestra en esta tabla:

Logical operators

Operator	*Definition*	*Example*
!	Not	! (Num1 < Num2)
&&	And	(Num1 < 5) && (Num2 > 10)
||	Or	(Num1 < 5) || (Num2 > 10)

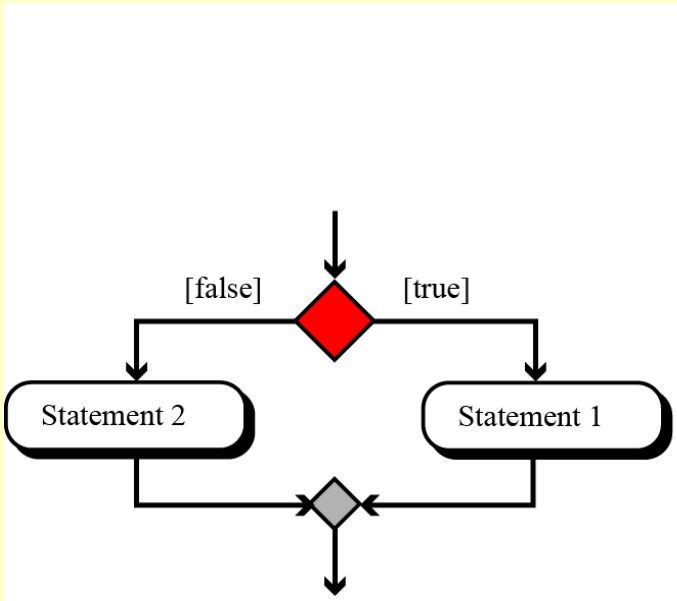
Declaraciones (sentencias)

Una declaración origina una acción a ser realizada por el programa. Lo traduce directamente en una o más instrucciones de computador ejecutables. Por ejemplo, C, C + + y Java definen varios tipos de declaraciones.

Una **sentencia/declaración de asignación** asigna un valor a una variable. En otras palabras, almacena el valor en la variable, que ya se ha creado en la sección de declaración.

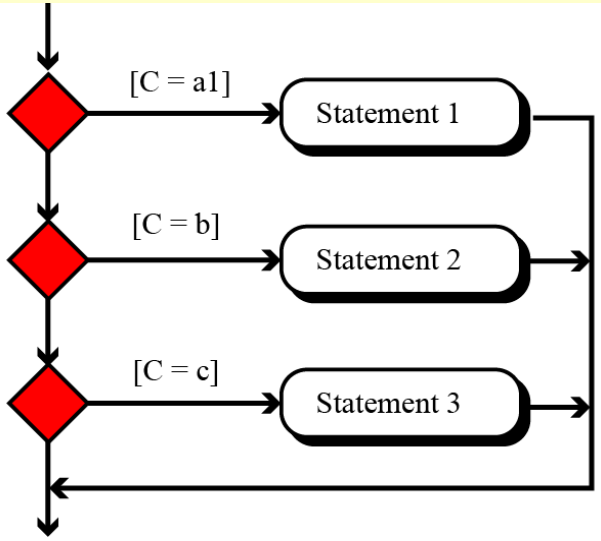
Una **sentencia/declaración compuesta** es una unidad de código que consta de cero o más declaraciones. También es conocido como un bloque. Una sentencia compuesta permite a un grupo de sentencias/declaraciones ser tratadas como una sola entidad.

La programación estructurada recomienda fuertemente el uso de los tres tipos de declaraciones de control: ***secuencia***, ***selección*** y ***repetición***, como ya discutimos en el capítulo anterior.




```
if (condition)
        statement1
else
        statement2;
```

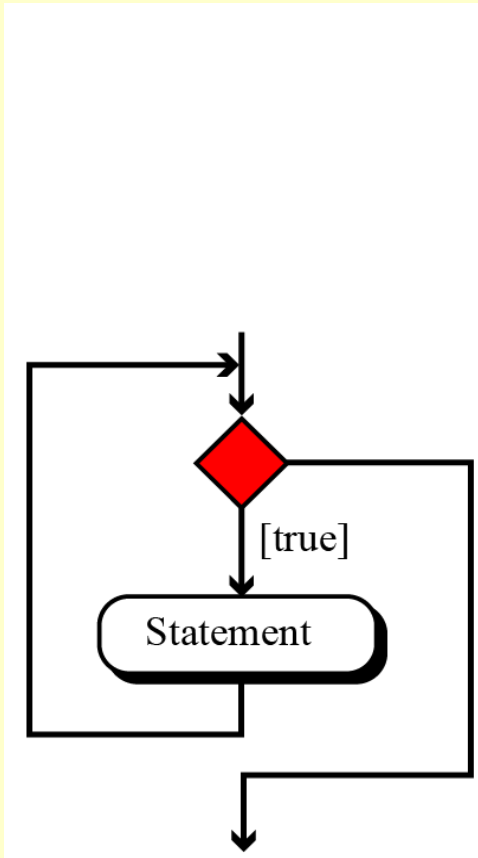
Code




```
switch (C)
 {
        case a: statement 1; break;
        case b: statement 2; break;
        case c: statement 3; break;
};
```

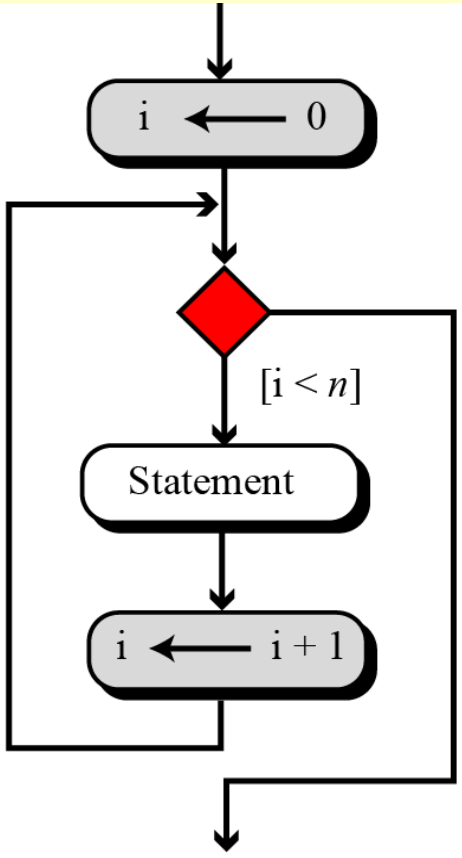
Code

Decisiones dobles y multi



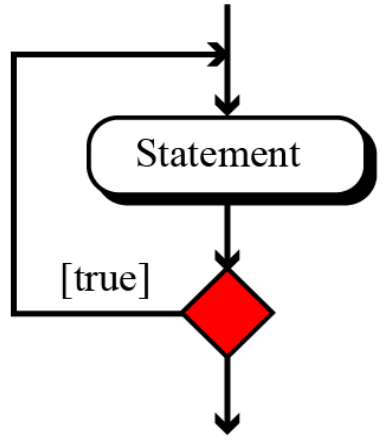

```
while (condition)
        statement
```

Code




```
for (int i = 0 ; i  <  n  ; i++)
        statement
```

Code



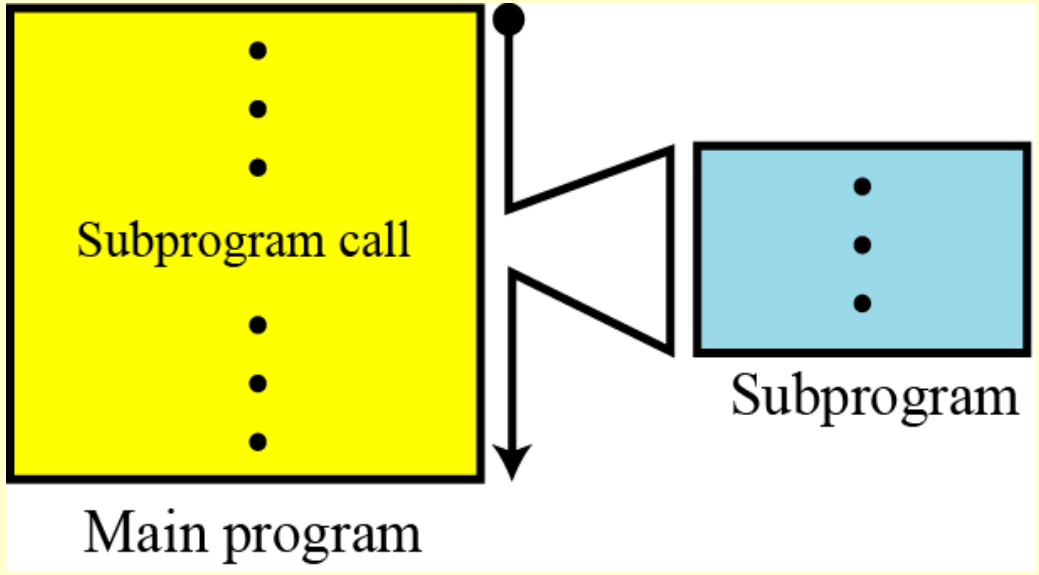

```
do
        statement
while (condition);
```

Code

Tres tipos de repeticion

Subprogramas

La idea de subprogramas es crucial en lenguajes de procedimiento y, en menor medida, en lenguajes orientados a objetos. Esto es útil porque el subprograma hace la programación más estructurada: un subprograma para realizar una tarea específica se puede escribir una vez, pero llamado muchas veces, al igual que los procedimientos predefinidos en el lenguaje de programación.



El concepto de un subprograma

En un lenguaje procedural, un subprograma, al igual que el programa principal, puede llamar a procedimientos predefinidos para operar en objetos locales. Estos objetos locales o **variables locales** son creadas cada vez que se llama al subprograma y destruidas cuando se devuelve el control del subprograma. Los objetos locales pertenecen a los subprogramas.

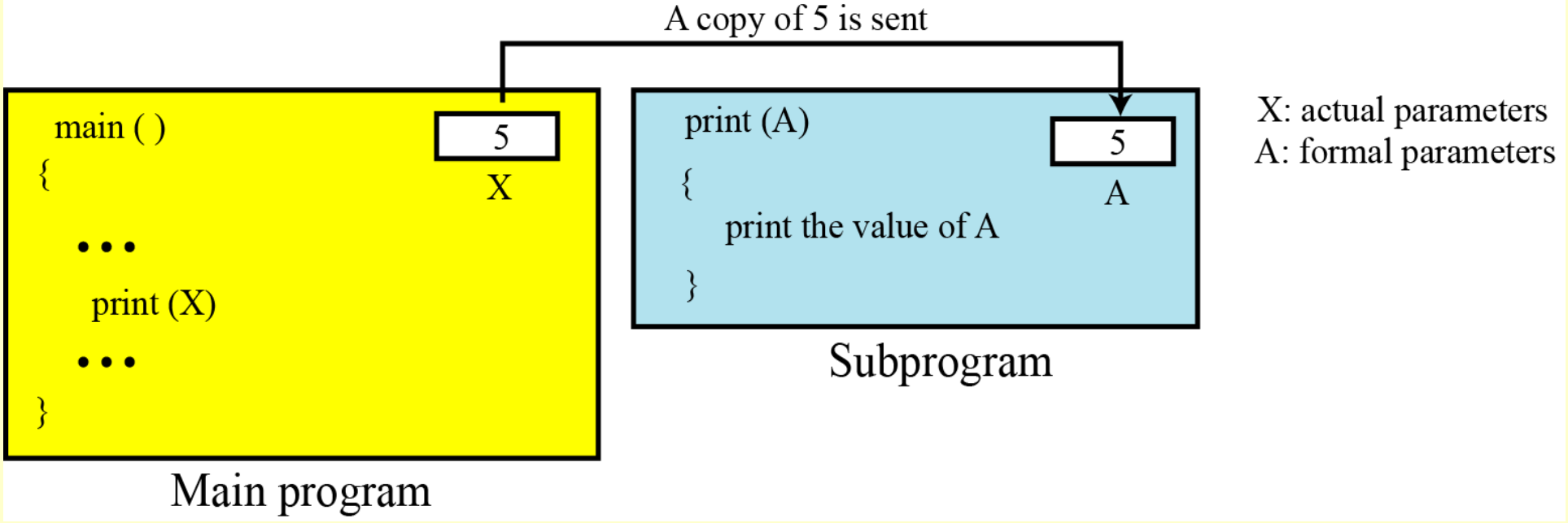
Es raro para un subprograma actuar sólo sobre objetos locales. La mayoría de las veces el programa principal requiere un subprograma para actuar sobre un objeto o conjunto de objetos creados por el programa principal. En este caso, el programa y subprograma usan **parámetros**. Estos se conocen como parámetros actuales en el programa principal y parámetros formales en el subprograma.

Paso por valor

En el paso de parámetro por valor, el programa principal y subprograma crean dos objetos (variables) diferentes. El objeto creado en el programa pertenece al programa y el objeto creado en el subprograma pertenece al subprograma. Ya que el territorio es diferente, los objetos correspondientes pueden tener el mismo nombre o nombres diferentes. La comunicación entre el programa principal y subprograma es de un solo sentido, desde el programa principal al subprograma.

Ejemplo 1

Supongamos que un subprograma es responsable de llevar a cabo la impresión para el programa principal. Cada vez que el programa principal quiere imprimir un valor, se lo envía al subprograma para su impresión. El programa principal tiene su propia variable X, el subprograma tiene su propia variable A. Lo que se envía desde el programa principal para el subprograma es el valor de la variable X.



Un ejemplo de paso por valor

Ejemplo 2

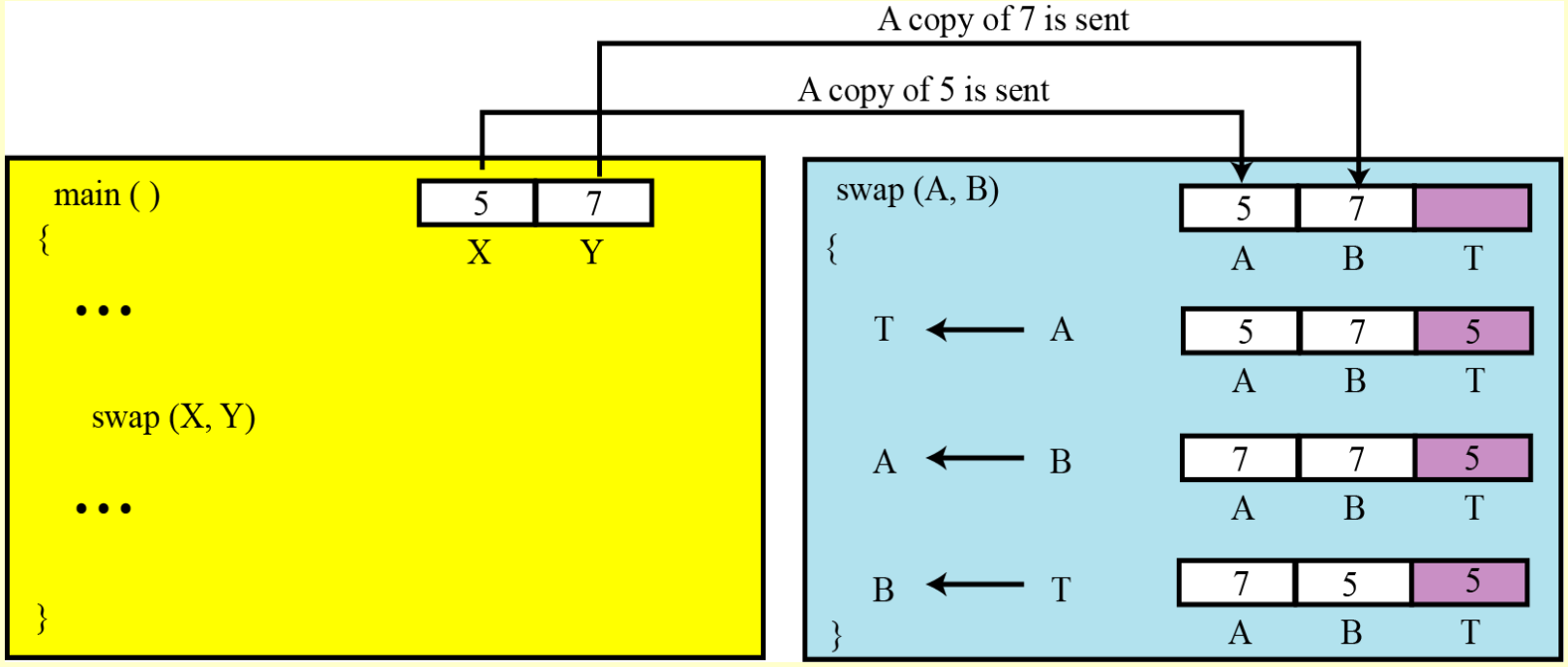
En el ejemplo 1, ya que el programa principal sólo envía un valor al subprograma, no es necesario tener una variable para este propósito: el programa principal puede enviar sólo un valor literal para el subprograma. En otras palabras, el programa principal puede llamar al subprograma de impresión / print(X) o print(5).

Ejemplo 3

Una analogía de paso por valor en la vida real es cuando un amigo quiere pedir prestado y leer un libro valioso que usted escribió. Puesto que el libro es precioso, posiblemente fuera de impresión, tu haces una copia del libro y se la pasas a tu amigo. Cualquier daño a la copia por lo tanto, no afecta al original.

Ejemplo 4

Supongamos que el programa principal tiene dos variables X e Y que tienen que intercambiar sus valores. El programa principal pasa para el subprograma los valores de X e Y, que son almacenados en dos variables A y B. El subprograma de intercambio utiliza una variable local T (temporal) e intercambia los dos valores A y B, pero los valores originales X e Y son los mismos: no se intercambian.



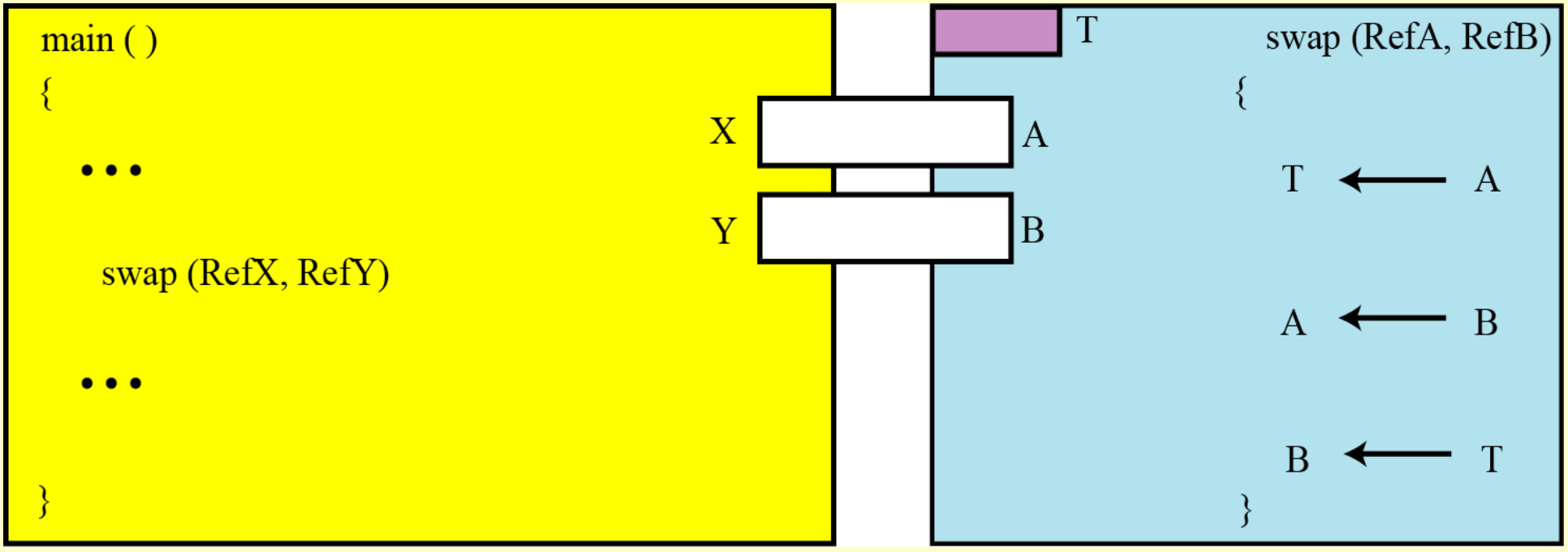
Un ejemplo en el que paso por valor no funciona

Paso por referencia

Paso por referencia fue ideado para permitir que un subprograma cambie el valor de una variable en el programa principal. En paso por referencia, la variable, que en realidad es una ubicacion en la memoria, es compartida por el programa principal y el subprograma. La misma variable puede tener diferentes nombres en el programa principal y el subprograma, pero ambos nombres se refieren a la misma variable. Metafóricamente hablando, podemos pensar en pasar por referencia como una caja con dos puertas: una se abre en el programa principal, la otra se abre en el subprograma. El programa principal puede dejar un valor en estea caja para el subprograma, el subprograma puede cambiar el valor original y dejar un nuevo valor para el programa en el mismo.

Ejemplo 5

Si utilizamos el mismo subprograma swap (de intercambio), pero dejamos que las variables se pasen por referencia, los dos valores de X e Y son realmente intercambiados.



Un ejemplo de paso por referencia

Valores de retorno

Un subprograma puede ser diseñado para devolver un valor o valores. Esta es la forma en que los procedimientos predefinidos están diseñados. Cuando utilizamos la expresión C ← A + B, en realidad llamamos a un proceso add(A, B) que devuelve un valor que se almacena en la variable C.