Práctica 8

1) Una sentencia simple es una instrucción **individual**, que no contiene otras sentencias en su interior. Realiza una única operación y **no requiere bloques adicionales**.
   Una sentencia compuesta está formada por **un grupo de sentencias**, que se ejecutan como una sola unidad. Se utilizan frecuentemente en estructuras de control como `if`, `while`, `for`, etc.

2) variable = expresión;
   - variable: es un identificador que debe haber sido previamente declarado.
   - expresión: puede ser un valor literal, otra variable, una operación aritmética, etc.

3) **Sí**, en C una expresión de asignación **puede producir efectos laterales** (*side effects*) que afectan el resultado final, especialmente cuando se involucran **operadores de incremento**, **llamadas a funciones**, o **expresiones múltiples** con orden no definido de evaluación.
   Ej:
   int a = 5;
   int b = a++ + a;
   ¿Qué valor toma b?
   - Depende del compilador, porque el orden en que se evalúan a++ y a no está definido por el estándar C.
   - Posibles valores: 10 o 11.

4) Un lenguaje utiliza **evaluación por circuito corto** cuando, al evaluar expresiones lógicas (`&&` o `||`), **se detiene tan pronto como el resultado de la expresión completa se puede determinar**.
   Ej: && → Si el primer operando es `false`, no evalúa el segundo.

   En un lenguaje con **circuito largo**, **todos los operandos se evalúan**, **sin importar el resultado parcial**. Esto **puede causar errores** si la segunda parte de la expresión depende de la primera (por ejemplo, división por cero o punteros nulos).
   Ej:
   ```
   int x = 0;
   if (x != 0 && 10 / x > 1) {
      printf("No se imprime\n");
   }
   ```

5) Todos ellos utilizan la regla del else más cercano:
   - El else se asocia con el if más cercano que no tenga else.

En C:
```
if (cond1)
   if (cond2)
      accion1;
   else
      accion2;   // ← se asocia con el if(cond2)
```

En Delphi:
```
if cond1 then
        if cond2 then
                accion1
        else
                accion2;  // ← también se asocia con el if cond2
```

En Ada: // es más estructurada y explícita: obliga a usar end if; para evitar ambigüedades.
```
if cond1 then
        if cond2 then
                accion1;
        else
                accion2;
        end if;
end if;
```

Python **usa la indentación** como delimitador de bloques. Esto elimina completamente la ambigüedad:
```
if cond1:
   if cond2:
      accion1
   else:
      accion2  # ← se asocia al if cond2 por la indentación
```

6) C usa la construcción `switch` para manejar **múltiples selecciones** basadas en el valor de una expresión entera o de tipo `char` (no acepta strings ni expresiones booleanas directamente).

```
switch (expresión) {
    case valor1:
        // instrucciones
        break;
    case valor2:
        // instrucciones
        break;
    default:
        // instrucciones si no coincide ningún caso
}
```

La expresión debe ser de tipo entero o char.
El break es necesario para evitar el "fall through" (caída al siguiente case).
El default es opcional, actúa como "else".

Lenguaje	Construcción	¿Requiere `break`?	¿Permite `fall through`?	¿Acepta tipos complejos?	¿Seguro contra errores?
C	`switch`	✅ Sí	✅ Sí	❌ Solo enteros y `char`	❌ No
Pascal	`case of`	❌ No	❌ No	❌ Solo enteros/enums	✅ Más seguro
Ada	`case`	❌ No	❌ No	❌ Solo tipos discretos	✅ Muy seguro
Python	`match` (3.10+)	❌ No	❌ No	✅ Sí (muy flexible)	✅ Muy seguro

7) En Pascal estándar:
El código puede traer problemas. El problema clave es:
   - Modificación de la variable de control del for dentro del bucle
   - En Pascal no está permitido modificar la variable de control del bucle for dentro del mismo cuerpo del bucle. En este caso, la procedure A modifica i con i := i + 1, lo que viola las reglas del lenguaje.

   En Ada estándar:
   También es inválido.
   - En Ada, la variable del bucle for es una constante dentro del bucle: no se puede modificar.
   - Intentar hacer i := i + 1; en Ada provocaría un error en tiempo de compilación.

   En Free Pascal / Turbo Pascal:
   - El compilador podría permitirlo, pero el comportamiento es incorrecto o indefinido.
   - La modificación de i podría no tener efecto, porque Pascal internamente controla el valor de i con su propio contador.

   En GNAT Ada:
   - No compila directamente.
   - Marca error: "cannot assign to loop parameter".

8) En Ada:
   Ada sí permite rangos en una sentencia case, y requiere cubrir todos los casos posibles (o usar when others).
   with Ada.Text_IO; use Ada.Text_IO;

```
procedure Evaluar_Puntos is
   puntos : Integer;
begin
   puntos := 3;  -- ejemplo de valor
   case puntos is
      when 1 .. 5 =>
         Put_Line("No puede continuar");
      when 10 =>
         Put_Line("Trabajo terminado");
      when others =>
         null;  -- caso por defecto obligatorio si no se cubren todos los posibles
   end case;
end Evaluar_Puntos;
```

En C:
El lenguaje C no permite rangos en switch, y tampoco permite múltiples valores por case de forma declarativa.

```
#include <stdio.h>
int main() {
   int puntos = 3;
   switch (puntos) {
      case 1:
      case 2:
      case 3:
      case 4:
      case 5:
         printf("No puede continuar\n");
         break;
      case 10:
         printf("Trabajo terminado\n");
         break;
      default:
         // ningún mensaje
         break;
   }
   return 0;
}
```

9)

Concepto	`return`	`yield`
Qué hace	Termina la función y devuelve un valor.	Pausa la función y devuelve un valor.

Estado	Pierde el estado al salir.	Retiene el estado y continúa desde donde se pausó.
Uso	Devuelve un resultado final.	Genera una secuencia de valores (generadores).
Memoria	Puede consumir mucha memoria si se devuelve una lista grande.	Eficiente para grandes volúmenes de datos (uno a uno).

¿Cuándo es útil yield?

- Cuando queremos procesar datos grandes o infinitos sin cargar todo en memoria.
- Por ejemplo: leer líneas de un archivo gigante, generar números primos, etc.

Usa return cuando necesitas un único valor o lista completa.
Usa yield cuando necesitas valores uno por uno (streaming) o eficiencia de memoria.

10) La instrucción `map` en JavaScript es un **método de los arrays** que permite **transformar** cada uno de sus elementos mediante una función, devolviendo **un nuevo array** con los resultados.

```
const nuevoArray = arrayOriginal.map((elemento, indice, array) => {
  // retornar valor transformado
});
```

`elemento`: valor actual del array que se está procesando.
`indice` *(opcional)*: índice del elemento.
`array` *(opcional)*: el array original.

Alternativas a map:

- **`for` tradicional**
  - Requiere más código y manejo manual del índice.
- **`forEach` + `push`**
  - Similar, pero no devuelve un array directamente.
- **`reduce`**
  - Más flexible, pero menos legible para este caso.
- **`Array.from()` con función de mapeo**
  - Poco común, pero válida.

11) Un **espacio de nombres** (namespace) es un mecanismo que permite **organizar y agrupar identificadores (nombres de clases, funciones, variables, etc.)** para **evitar conflictos** cuando distintos elementos tienen el mismo nombre.
Java implementa espacios de nombres mediante el uso de **paquetes** (`packages`).
A través de la palabra clave `package`, que define el espacio de nombres de una clase.

Con `import`, se puede acceder a clases que están en otros paquetes.
Ej:

```
package com.ejemplo.util;
public class Utilidad {
    public static void saludar() {
        System.out.println("¡Hola!");
    }
}

import com.ejemplo.util.Utilidad;
public class Main {
    public static void main(String[] args) {
        Utilidad.saludar();
    }
}
```

Python

- Usa módulos y paquetes.
- Cada archivo .py representa un espacio de nombres.
- Se puede importar parcial o totalmente.

Tiene **espacios de nombres locales, globales y built-in**, y su resolución sigue el orden **LEGB** (Local, Enclosing, Global, Built-in).

PHP

- Implementa espacios de nombres desde PHP 5.3 con namespace.
- Permite definir funciones, clases o constantes en distintos espacios de nombres aunque se llamen igual.
- Utiliza use para alias y resolución de conflictos.