Conceptos y Paradigmas de Lenguajes de Programación - Práctica 8

EJERCICIO 1

Sentencia Simple
Es una expresión o combinación de expresiones que realiza una acción específica, como asignar un valor a una variable, realizar una operación aritmética, llamar a una función o controlar el flujo del programa. Las sentencias suelen terminar con ";" en muchos lenguajes de programación.

En otras palabras, es una instrucción individual que realiza una sola acción. No contiene otras sentencias dentro de ella.
Ejemplo en Java: **x = 5;**
Esta sentencia asigna el valor 5 a una variable llamada x.

Sentencia Compuesta
Es un conjunto de sentencias agrupadas que se ejecutan secuencialmente.
En muchos lenguajes de programación, las sentencias compuestas se encierran entre llaves (0) para definir un bloque. Las sentencias compuestas permiten agrupar varias sentencias como si fueran una sola, lo cual es útil en estructuras de control como if, for, while, y en la definición de funciones.

EJERCICIO 2

La **asignación en C** es una de las operaciones más fundamentales del lenguaje, y se implementa de forma directa, con un operador sencillo y eficiente. Se realiza con el operador =, que copia el valor del lado derecho (expresión) al lado izquierdo (variable). Es decir, de izquierda a derecha. Si se usa asignaciones en condiciones primero asigna y luego evalúa.

Ejemplos:
A=B=C=0 → 0 se devuelve como r-valor a todos, primero se asigna a C, luego C asigna a B, luego B asigna a A, y todos valen 0.
if (i=30) → no es una comparación, ya que C compara con ==, primero asigna el valor 30 a la variable i, luego interpreta que i es distinto de 0 y da verdadero -> **toma como verdadero todo resultado distinto de 0.**

EJERCICIO 3

Si, una expresión de asignación puede producir efectos colaterales que afectan al resultado final, como es el caso del ejemplo anterior en C.

EJERCICIO 4

Circuito corto
- En un lenguaje que utiliza el circuito corto, la evaluación de una expresión lógica se detiene tan pronto como se determina el resultado final sin necesidad de evaluar el resto de la expresión.
- También conocida como evaluación perezosa o evaluación de cortocircuito.

- **Operador (“y”, “and”, “&&”):** Da como resultado verdadero únicamente cuando ambos términos son verdaderos. Si el primer término es falso, no es necesario evaluar el segundo ya que el resultado será falso.
- **Operador(“o” / “or” / “||”):** Da como resultado Falso únicamente cuando ambos términos son falsos. Si el primer término es verdadero, no es necesario evaluar el segundo ya que el resultado será verdadero.
- Permite evitar errores y optimizar el rendimiento.
- Permite evitar errores al evaluar expresiones indefinidas, es decir, si existe una condición “A and B” en la que “B” es nulo, se puede dar un error al evaluarla. Entonces podemos hacer que la expresión “A” garantice que “B” no sea nulo.

Circuito largo
- En un lenguaje que utiliza el circuito largo, la evaluación de una expresión lógica continúa evaluando todas las partes de la expresión, incluso si el resultado final ya se ha determinado.También conocida como evaluación perezosa o evaluación de cortocircuito.
- **Operador (“y”, “and”, “&&”):** ambas partes de la expresión se evalúan independientemente del valor de la primera parte. Esto asegura que se realicen todas las comprobaciones necesarias.
- **Operador(“o” / “or” / “||”):** ambas partes de la expresión se evalúan independientemente del valor de la primera parte. Esto asegura que se realicen todas las comprobaciones necesarias.

Ejemplo en C, que NO da error en circuito corto, pero si da error en circuito largo:

```c
int a = 0;
if (a != 0 && (10 / a) > 1) {
    printf("Verdadero\n");
}
```

a !=0 → Al ser falso, en circuito corto no evaluará el segundo término, que es una división por cero, la cual sí será evaluada en circuito largo y producirá un error.

EJERCICIO 5

C: como regla define que cada rama else se empareja con la instrucción if solitaria más próxima, es decir, cada else se empareja para cerrar al último if abierto.

Ada y Delphi: utilizan la coincidencia de las palabras clave begin y end para determinar la asociación del else con el if correspondiente más cercano.

Python: hace uso de la indentación para determinar el cuerpo de las estructuras if, elif y del else, por lo tanto, la asociación el else se asocia con el if anterior que tenga la misma indentación.

EJERCICIO 6

SELECCIÓN MÚLTIPLE EN C

- Constructor **Switch (expresión)**
- **Cada rama Case** es **"etiquetada"** por uno o más **valores constantes (*enteros o char*)**
- Si **coincide** con una **etiqueta** se **ejecutan las sentencias asociadas, y continúa con las sentencias de las otras entradas**. (*chequea todas salvo exista un **break***)
- La **sentencia break (opcional)** provoca la **salida de cada rama**
- La **sentencia default (opcional)** para los casos en que el **valor no coincida con ninguna de las opciones establecidas**
- El **orden** en que aparecen las ramas **no tiene importancia**

38

```
Switch (Operador) {
    case  '+' :
            result:= a + b; break;
    case  '-':
            result:= a - b; break;
    default : //Opcional
            result:= a * b;
    }
```

Conviene usar **break** para saltar las siguientes ramas, **sinó pasa por todas**

```
switch( i )
{
   case -1:
     n++;
     break;
   case 0 :
     z++;
     break;
   case 1 :
     p++;
     break;
}
```

Para **evitar problemas** y **asegurar comportamiento**, es **buena práctica** siempre **tener en cuenta todas las posibles entradas** que puede recibir la expresión del switch y manejarlas adecuadamente **con case específicos y un default** si es necesario.

41

SELECCIÓN MÚLTIPLE EN PASCAL

- Usa **palabra reservada case** seguida de **variable-expresión de tipo ordinal** y la **palabra reservada of**.
- **Variable-expresión** a evaluar es llamada "**selector**"
- **Lista de sentencias** para los diferentes valores que puede adoptar la variable **(los "casos"), que además llevan etiquetas.**
- **No importa el orden**
- **bloque else** para el caso que la variable adopte un valor que **no coincida con ninguna de las sentencias de la lista. (opcional)**
- Para **finalizar** se coloca un **end;** (no se corresponde con ningún "begin" que exista).

Inseguro: al no establecer que sucede cuando un valor no cae dentro de ninguna de las alternativas.

Sintaxis:

```
case variable_ordinal of
      valor1: sentencia 1;
      valor2: sentencia2;
      valor3: sentencia3;
else
      sentencia4;
    end;
```

Ordinal: puede obtenerse un predecesor y un sucesor (a excepción del primer y el último (expresa la idea de orden o sucesión)

32

else es opcional. ---- Inseguro

¿Qué sucede si se ingresa un número y no hay un else?

- La **"no acción"** puede **ocultar errores lógicos graves, comportamientos inesperados**, variables sin definir, funciones sin ejecutar, etc.
- **Algunos compiladores no advierten si no se cubren todos los casos posibles**
- **El programador debe ocuparse de considerar** todas las **opciones**

Ejemplo:

```
var opcion : char;
begin
  readln(opcion);
case opcion of
     '1' : nuevaEntrada;
     '2' : cambiarDatos;
     '3' : borrarEntrada
     else
      writeln('Opcion no valida!!')
     end;
end
```

Doble end por el **begin que ya existe**

else cambió entre versiones de Pascal,

otherwise

33

SELECCIÓN MÚLTIPLE EN ADA

Constructor:
Case expresión **is**
con **when** y **end case**

- Tiene **reglas** más **estrictas**.
- Las **expresiones sólo** pueden ser de **tipo entero o enumerativo**
- El **case debe contemplar todos los valores posibles** que puede tomar la expresión. ***Sino error del compilador***
- El **when =>** para **indicar la acción/ sentencia a ejecutar si se cumple la condición.**
- **end case; debe** ser siempre el **cierre final**

34

- **When Others** se puede utilizar para **representar a aquellos valores que no se especificaron explícitamente. (opcional)**
- **When Others debe ser** la **última opción antes del end case;**
- Una vez que **se ejecuta una rama del case,** el **case finaliza y no se ejecuta ninguna otra rama.**

No pasa la compilación si:

NO se coloca la rama para un posible valor y

NO aparece la opción Others en esos casos

35

Ejemplo 1

```
case Operador is
      when  '+' => result:= a + b;
      when  '-' => result:= a - b;
      when others => result:= a * b;
end case;
```

Importante para el programador:

others se debe colocar porque las etiquetas de las ramas NO abarcan todos los posibles valores de Operador

Ejemplo 2

```
type Dia_Semana is (Lunes, Martes, Miercoles);
Dia : Dia_Semana;

case Dia is
  when Lunes => ...
  when Martes => ...
end case;
```

ERROR compilación x miércoles

SELECCIÓN MÚLTIPLE EN PYTHON

- match seguido de la expresión a evaluar.
- case seguido del valor a comparar.
- (guión bajo) se utiliza como comodín para manejar casos no especificados.
- No se necesita break ya que cada caso es inherentemente separado y explícito.

```
match variable:
    case 1:
        accion1()
    case 2:
        accion2()
    case _:
        accion_por_defecto()
```

EJERCICIO 7-A

Pascal: En el caso específico de Pascal el código no genera error aunque conceptualmente esté mal ya que modifica manualmente la variable iteradora, lo que hace que no se genere error es que la modificación de la variable no se da dentro de la misma unidad que el bloque for, si esto hubiera sido así, se generaría el error “Error: Illegal assignment to for-loop variable ‘i’”.

ADA: En el caso de Ada el código si generaría error ya que la variable de control de los for en Ada se declara implícitamente al entrar al bucle y desaparece al salir del mismo, eso quiere decir que cuando se intente modificar el valor de “i” en el procedimiento, la variable no tendrá valor.

EJERCICIO 7-B

El comportamiento en otras versiones será el mismo en ambos casos.

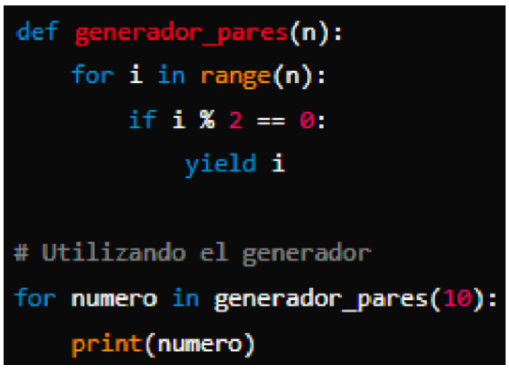
EJERCICIO 8

C y Pascal: Ese código en C y Pascal no genera error ya que en esos lenguajes si una variable no cae dentro de alguno de los casos de la lista de casos del case, se sigue la ejecución y la variable no se verá modificada, lo que sí puede llegar a ocurrir es que se presenten efectos colaterales en el código si no se chequea o maneja este tipo de situaciones.

ADA: En Ada, ese código si es trasladado sin un bloque Others genera un error en compilación ya que no se están considerando todos los casos posibles para el case, en cambio si se trasladara con un bloque Others entonces no habría problemas ya que si la variable no cae en un caso de la lista de casos, entonces será ejecutado el bloque Others con la lógica correspondiente para tratar esa situación.

EJERCICIO 9

Característica	`return`	`yield`
Finaliza la función	✅ Sí (termina y devuelve un valor)	❌ No (pausa y puede continuar después)
Devuelve	Un único valor	Un **generador** (secuencia de valores)
Uso principal	Funciones tradicionales	**Generadores** (iteración perezosa)
Memoria	Carga todo en memoria	Genera elementos uno a uno (más eficiente)

Yield se vuelve útil cuando se necesita producir una secuencia de valores uno a la vez y mantener el estado de la función entre las llamadas. Podemos ver por ejemplo una función que calcule números pares hasta el número “n”. En lugar de calcular todos los números pares hasta un límite y almacenarlos en una lista, podemos utilizar un generador yield para obtener los números pares uno a uno, evitando así almacenar todos los números en memoria.	

EJERCICIO 10

La función map en JavaScript es una función de orden superior que se utiliza para transformar los elementos de un arreglo original y generar un nuevo arreglo con los resultados de aplicar una función a cada elemento del arreglo original.

Sus alternativas son:

- **forEach**: itera sobre cada elemento de un array y ejecuta una función para cada elemento, pero no devuelve un nuevo array, modificar el original.
- **Filter**: crea un nuevo array con todos los elementos que pasan por un cierto Filtro.
- **Reduce**: reduce un array a un solo valor aplicando una función a cada elemento y acumulando el resultado.
- **FlatMap**: similar al map, pero primero mapea cada elemento usando una función de mapeo, luego aplana los resultados en un solo array.

EJERCICIO 11

En **Python**, el manejo de espacio de nombres se implementa utilizando diferentes estructuras de ámbito (scope), que determinan dónde se pueden utilizar y acceder a los nombres. Las características más importantes del manejo de espacio de nombres en Python incluyen:

Scope local y global	Python utiliza un scope local para cada función, donde las variables definidas dentro de la función solo son accesibles dentro de ella. Además, hay un scope global donde las variables definidas fuera de cualquier función son accesibles en todo el módulo.
Scope de módulo	Cada módulo en Python tiene su propio espacio de nombres global, lo que permite que las variables, funciones y clases definidas en un módulo sean accesibles sólo dentro de ese módulo, a menos que sean exportadas explícitamente.
Scope de clase	En Python, las clases también tienen su propio espacio de nombres, lo que permite que los atributos y métodos de una clase se definan y accedan de forma independiente a otras clases.
Namespaces anidados	Python admite la anidación de espacios de nombres, lo que significa que se pueden tener espacios de nombres dentro de otros. Por ejemplo, dentro de una función, puede haber un namespace local, pero también se puede acceder a los namespaces globales y de módulo.