Práctica 9 Estructuras de control y sentencias

> **`Objetivo`** reconocer las diferencias entre las implementaciones de las estructuras de control de los distintos lenguajes

Ejercicio 1

Una sentencia puede ser simple o compuesta, ¿Cuál es la diferencia?

La diferencia entre una sentencia **simple** y una **sentencia** compuesta radica en la cantidad de acciones o instrucciones que se agrupan en una única unidad. Una sentencia simple consta de una única acción, mientras que una sentencia compuesta agrupa varias sentencias en un bloque de código delimitado por llaves.

Una sentencia simple consta de una única acción o instrucción que se ejecuta secuencialmente. Esta acción puede ser una asignación, una llamada a una función, una expresión, un retorno, etc. Ejemplos de sentencias simples pueden ser:

- Asignación: **`x = 5;`**
- Llamada a función: **`print("Hola");`**
- Expresión: **`y = x + 3;`**
- Retorno: **`return resultado;`**

Por otro lado, una sentencia compuesta, también conocida como bloque de código, está formada por un conjunto de sentencias agrupadas entre llaves `{}` o algún otro delimitador. Estas sentencias se ejecutan en secuencia y se consideran como una sola unidad. Ejemplos de sentencias compuestas son:

```
if (x > 0) {
   System.out.println("El valor es positivo");
   x = x * 2;
}
```

En este ejemplo, las sentencias **`System.out.println("El valor es positivo");`** y **`x = x * 2;`** están agrupadas en un bloque de código que se ejecuta si la condición **`x > 0`** es verdadera.

Ejercicio 2

Analice como C implementa la asignación.

En el lenguaje de programación C, la asignación se implementa mediante el operador de asignación (`=`). Se evalúa una expresión en el lado *derecho* y se copia el valor resultante en la variable del lado *izquierdo*. La asignación se basa en el tipo de dato de la variable y no establece una relación de referencia.

```
int a = 5;
int b;
b = a + 3; // Asignación de la expresión a la variable b
```

Ejercicio 3

¿Una expresión de asignación puede producir efectos laterales que afecten al resultado final, dependiendo de cómo se evalúe?

De ejemplos.

Sí, una expresión de asignación en algunos lenguajes de programación puede tener efectos laterales que afecten al resultado final, dependiendo de cómo se evalúe. Un efecto lateral es cualquier cambio o acción que se produce además de la asignación del valor a la variable.

Un ejemplo de efecto lateral en una expresión de asignación en C es el operador de incremento (`++`) o decremento (`--`). Estos operadores modifican el valor de la variable y, a su vez, devuelven el valor modificado. Dependiendo de si se usan antes o después de la variable, pueden tener un efecto lateral en el resultado final. En este ejemplo, se asigna el valor de `a` a `b` utilizando el operador de incremento posterior (`a++`). El efecto lateral es que `a` se incrementa después de la asignación, por lo que `b` recibe el valor original de `a` antes del incremento.

```
int a = 5;
int b = a++; // Efecto lateral: incrementa a después de asignar a b

printf("a: %d\n", a); // Salida: 6
printf("b: %d\n", b); // Salida: 5
```

Otro ejemplo podría ser el siguiente, donde si se lee de izquierda a derecha, imprime `3 3 3` , tomando el valor del último valor. Si se lee de derecha a izquierda, imprime `1 1 1` , toman el ultimo valor es decir el de a.

```c
int main()
{
   int a = 1;
   int b = 2;
   int c = 3;
   a = b = c;
   printf("%d\n", a);
   printf("%d\n", b);
   printf("%d\n", c);
}
```

Otro ejemplo podría ser una función con parámetros, si se lee al revés (de derecha a izquierda) y la ligadura es posicional: la posición 2 es la 1 y la 1 es la 2.

Ejercicio 4

Qué significa que un lenguaje utilice circuito corto o circuito largo para la evaluación de una expresión.

De un ejemplo en el cual por un circuito de error y por el otro no.

Las condiciones de estas sentencias de selección tienen dos formas de analizarse a nivel de código:

- ***Circuito corto***: Evalúa la expresión lógica hasta encontrar un `false`, cuando lo encuentre termina la evaluación y devuelve `false`.
  - Si es un “AND”, si una de las condiciones da falsa, no evalúa las siguientes y retorna falso.
  - Si es “OR” , basta con que una condición sea verdadera para retornar verdadero, en dicho caso no evalúa las siguientes.
- ***Circuito largo***: Evalúa toda la expresión sin importar lo que se encuentre, al final devuelve el resultado de toda la expresión lógica.

En el siguiente ejemplo podemos observar la comparación de ambos. En Java se puede elegir qué tipo de circuito usar al evaluar una condición lógica. Por ejemplo `&&` es bajo circuito corto y `&` es circuito largo.

```c
int a = 1;
Object objecto = null;

// Circuito corto: La evaluación se detiene en la primera parte falsa
if (a == 0 && objecto == null) {
   System.out.println("sale a la primera evaluación");
}

// Circuito largo: Se evalúan ambas partes de la expresión
if (a == 0 & objecto.toString().equals('algo')) {
   System.out.println("esto va a tirar error!!");
}
```

Ejercicio 5

¿Qué regla define Delphi, Ada y C para la asociación del else con el if correspondiente?

Delphi, Ada y C utilizan una regla conocida como "regla del else ambiguo" para la asociación del `else` con el `if` correspondiente. Esta regla establece que el `else` se asociará con el `if` más cercano y no resuelto en el código.

En Delphi y Ada, esta regla se aplica de manera estricta, lo que significa que el `else` se asociará solo con el `if` más cercano y no se permiten ambigüedades. Si hay múltiples `if` anidados, cada `else` se asociará con el `if` inmediatamente anterior en su ámbito más cercano y no se permitirá otra interpretación.

En C, la regla del "else ambiguo" también se aplica, pero de manera menos estricta. En C, si no se utilizan llaves (`{}`) para delimitar el alcance de los bloques de código de los `if` y `else` , el `else` se asociará con el `if` más cercano que no tiene un `else` correspondiente. Esto puede llevar a ambigüedades y problemas si no se utilizan las llaves de manera adecuada.

¿Cómo lo maneja Python?

Por otro lado, Python no utiliza la regla del "else ambiguo" como Delphi, Ada y C. En Python, la asociación del `else` se determina por la indentación o sangría. El `else` se asocia con el `if` correspondiente basándose en la alineación de indentación. Esto significa que la estructura del código en Python se basa en la indentación y no en reglas de asociación específicas. Además, el if de Python es sin ambigüedad y legible (incorpora `elif` y sangría).

En resumen, Delphi, Ada y C siguen la regla del "else ambiguo" para asociar el `else` con el `if` más cercano y no resuelto. En cambio, Python utiliza la indentación para determinar la asociación del `else` con el `if` correspondiente.

Ejercicio 6

¿Cuál es la construcción para expresar múltiples selección que implementa C?

En C, la construcción para expresar múltiples selecciones es el uso de la instrucción `switch` . La instrucción `switch` permite evaluar una expresión y ejecutar diferentes bloques de código según el valor de la expresión. En C, la estructura de la instrucción `switch` es la siguiente:

```
switch (expresion) {
   case valor1:
      // Código a ejecutar si expresion == valor1
      break;
   case valor2:
      // Código a ejecutar si expresion == valor2
      break;
   // Otros casos posibles
   default:
      // Código a ejecutar si no se cumple ningún caso anterior
      break;
}
```

El flujo de ejecución dentro de un **`switch`** depende del valor de la expresión. Si el valor de la expresión coincide con uno de los casos (**`valor1`** , **`valor2`** , etc.), se ejecutará el bloque de código correspondiente a ese caso y luego se saldrá del **`switch`** mediante la instrucción **`break`** . Si el valor de la expresión no coincide con ninguno de los casos, se ejecutará el bloque de código dentro del **`default`** .

¿Trabaja de la misma manera que la de Pascal, ADA o Python?

La construcción **`switch`** en C funciona de manera similar a la de Pascal y ADA, ya que evalúa una expresión y ejecuta el bloque de código correspondiente al valor de la expresión. Sin embargo, hay algunas diferencias en la sintaxis y en el manejo de los casos.

Pascal

bloque `else` para el caso que la variable adopte un valor que no coincida con ninguna de las sentencias de la lista. (opcional)

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

ADA

Tiene la cláusula `Others` que se puede utilizar para representar a aquellos valores que no se especificaron explícitamente.

```
case Hoy is
   when MIE..VIE => Entrenar_duro; -- Se puede especificar Rango con ..
   when MAR |SAB => Entrenar_poco; -- Se puede especificar varias elecciones |
   when DOM => Competir; -- Única elección.
   when others => Descansar; -- Debe ser única y la última alternativa.
   end
case;
```

En Python, no existe una construcción directa equivalente al **`switch`** de C. En su lugar, se suele utilizar una serie de declaraciones **`if-elif-else`** para expresar múltiples selecciones en Python.

Ejercicio 7

Sea el siguiente código

```
.....
var i, z:integer;
Procedure A;
begin
  i:= i +1;
end;
begin
  z:=5
```

```
  for i:=1..5 do
begin
  z:=z*5;
  A;
  z:=z + i;
end;
end;
```

a) Analice en las versiones estándar de ADA y Pascal, si este código puede llegar a traer problemas. Justifique la respuesta.

Pascal estándar "no permite" que se modifiquen los valores del límite inferior, límite superior, ni del valor de la variable de control. Como el procedimiento `A` modifica la variable de control esto hará efecto colateral (o el error: `Error: Illegal assignment to for-loop variable "i"` , no estoy segura //CONSULTAR).

En ADA la variable del `for` es considerada como variable local, por lo cual en A no se modifica.

b) Comente qué sucedería con las versiones de Pascal y ADA, que Ud. utilizó. 🥳

Ejercicio 8

Sea el siguiente código en Pascal

```
var puntos: integer;
begin
  ...
  case puntos
  1..5: write(“No puede continuar”);
  10:write(“Trabajo terminado”)
end;
..
```

Analice, si esto mismo, con la sintaxis correspondiente, puede trasladarse así a los lenguajes ADA, C. ¿Provocaría error en algún caso? Diga cómo debería hacerse en cada lenguaje y explique el por qué. Codifíquelo.

```
var puntos: integer;
begin
case puntos
   1..5: write(“No puede continuar”);
   10:write(“Trabajo terminado”)
end;
```

En el caso de Ada, la estructura `case` es conocida como `case statement` . Para trasladar el código a Ada, se debe utilizar la sintaxis adecuada y tener en cuenta algunas diferencias:

```
puntos: integer;
begin
...
case puntos is
   when 1..5 =>
      put("No puede continuar");
   when 10 =>
      put("Trabajo terminado");
end case;
...
```

En Ada, se utiliza `is` después de `case puntos` para indicar que se está realizando una comparación. Además, se utiliza `=>` en lugar de `:` para especificar la acción a realizar cuando se cumple cada caso.

En el caso de C, la estructura equivalente es el `switch`. Sin embargo, en C, la sintaxis de `case` no permite rangos directamente. Por lo tanto, se deben utilizar sentencias individuales `case` para cada valor o combinar múltiples `case`:

```
int puntos;
...
switch (puntos) {
   case 1:
   case 2:
   case 3:
   case 4:
   case 5:
      printf("No puede continuar");
      break;
   case 10:
      printf("Trabajo terminado");
      break;
   default:
      // Acción por defecto
      break;
}
...
```

En este caso, se enumeran los casos individuales de 1 a 5 para que tengan el mismo comportamiento. La sentencia `break` se utiliza para salir del `switch` después de cada caso. La sección `default` se utiliza para especificar una acción por defecto en caso de que ninguno de los casos coincida.

Ejercicio 9

Qué diferencia existe entre el generador YIELD de Python y el return de una función. De un ejemplo donde sería útil utilizarlo.

La diferencia principal entre el generador `yield` de Python y la declaración `return` en una función es que `yield` permite la generación de valores de forma iterativa, mientras que `return` finaliza la ejecución de una función y devuelve un valor final.

Cuando se utiliza `yield` en una función, esta se convierte en un generador. El generador puede pausar su ejecución en cada iteración y generar un valor que se devuelve al iterador que lo llamó. Luego, en la siguiente

iteración, el generador continúa su ejecución a partir del punto donde se detuvo y produce el siguiente valor. Este proceso de pausa y reanudación puede ocurrir varias veces hasta que se agoten los valores o se alcance una condición de finalización.

En cambio, la declaración `return` se utiliza para devolver un valor específico y finalizar la ejecución de la función. Cuando se encuentra una instrucción `return`, la función deja de ejecutarse y devuelve el valor indicado. No es posible reanudar la ejecución de una función después de la declaración `return`.

Un ejemplo donde sería útil utilizar `yield` en lugar de `return` es al implementar un generador que produce una secuencia infinita de números pares. En lugar de calcular todos los números pares de antemano y almacenarlos en una lista, podemos usar un generador para producir los números de forma incremental:

```
def numeros_pares_infinitos():
      num = 0
      while True:
         yield num
         num += 2

generador = numeros_pares_infinitos()

# Obtener los primeros 5 números pares
for _ in range(5):
      print(next(generador))
```

En este ejemplo, el generador `numeros_pares_infinitos` utiliza `yield` para generar números pares de forma infinita. Cada vez que se llama a `next(generador)`, se pausa la ejecución del generador, se devuelve el número par actual y luego se reanuda en la siguiente iteración. Esto permite obtener una secuencia infinita de números pares sin necesidad de almacenarlos todos en memoria.

Ejercicio 10

Describa brevemente la instrucción map en javascript y sus alternativas.

La instrucción `map` en JavaScript es una función que se utiliza para transformar los elementos de un array. Toma como argumento una función de transformación y devuelve un nuevo array con los resultados de aplicar esa función a cada elemento del array original, en el mismo orden. La sintaxis básica del `map` es la siguiente: `const newArray = array.map(funcionDeTransformacion);` La función `funcionDeTransformacion` se aplica a cada elemento del array `array`, y el resultado se agrega al nuevo array `newArray`. El nuevo array tendrá la misma longitud que el original.

Alternativas a la instrucción `map` en JavaScript incluyen:

1. Ciclo `for`: Puedes utilizar un ciclo `for` tradicional para iterar sobre los elementos del array y realizar la transformación manualmente. Esto implica crear un nuevo array vacío y agregar los resultados de la transformación uno por uno en cada iteración del ciclo.

```
const newArray = [];
for (let i = 0; i < array.length; i++) {
const transformedValue = funcionDeTransformacion(array[i]);
newArray.push(transformedValue);
}
```

2. **`forEach`** : El método **`forEach`** permite iterar sobre los elementos de un array y ejecutar una función en cada elemento, pero no devuelve un nuevo array con los resultados de la transformación. En su lugar, se utiliza para realizar operaciones en cada elemento sin modificar el array original.

```
const newArray = [];
array.forEach((element) => {
const transformedValue = funcionDeTransformacion(element);
newArray.push(transformedValue);
});
```

Estas alternativas pueden ser útiles cuando se desea mayor control o flexibilidad en el proceso de transformación de los elementos del array. Sin embargo, **`map`** proporciona una forma más concisa y expresiva de realizar transformaciones en los elementos de un array, lo que lo hace ampliamente utilizado y preferido en muchos casos.

Ejercicio 11

Determine si el lenguaje que utiliza frecuentemente implementa instrucciones para el manejo de espacio de nombres. Mencione brevemente qué significa este concepto y enuncie la forma en que su lenguaje lo implementa. Enuncie las características más importantes de este concepto en lenguajes como PHP o Python.

JavaScript implementa instrucciones para el manejo de espacio de nombres a través del uso de objetos y la estructura de ámbito léxico. En JavaScript, cada objeto actúa como un espacio de nombres que puede contener variables, funciones y otros objetos.

El concepto de espacio de nombres en JavaScript se refiere a la capacidad de tener identificadores únicos para evitar conflictos entre variables y funciones con el mismo nombre. Permite organizar y estructurar el código para evitar colisiones de nombres y mantener una buena modularidad.

En JavaScript, se pueden crear objetos para actuar como espacios de nombres utilizando la notación de objetos literales, por ejemplo:

```
const miNamespace = {
variable1: "valor1",
variable2: "valor2",
funcion1: function () {
   // Código de la función 1
},
funcion2: function () {
   // Código de la función 2
},
};
```

En este ejemplo, **`miNamespace`** es un objeto que actúa como espacio de nombres y contiene variables y funciones.

Al utilizar espacios de nombres en JavaScript, podemos evitar colisiones de nombres entre diferentes partes de nuestro código y organizar mejor nuestras variables y funciones. Esto es especialmente útil cuando trabajamos en

proyectos grandes y colaborativos, donde varios desarrolladores pueden estar escribiendo código al mismo tiempo.

En cuanto a las características importantes del manejo de espacio de nombres en otros lenguajes:

- **PHP** En PHP, el manejo de espacio de nombres se logra mediante la declaración `namespace`. Permite organizar las clases, funciones y constantes en espacios de nombres distintos, evitando conflictos entre ellos.
- **Python** En Python, el manejo de espacio de nombres se realiza a través de los módulos y paquetes. Los módulos actúan como espacios de nombres y contienen variables, funciones y clases. Los paquetes son carpetas que contienen módulos relacionados y también actúan como espacios de nombres. El uso de `import` y `from` en Python permite acceder a los elementos definidos en diferentes espacios de nombres.