Práctica 8: Estructuras de control y sentencias

Ejercicio 1: Una sentencia puede ser simple o compuesta, ¿Cuál es la diferencia?

Sentencia simple: Es una instrucción básica que realiza una sola acción y no contiene otras sentencias anidadas. Generalmente, se ejecuta de manera secuencial y corresponde a una línea de código que completa una tarea específica.

Sentencia compuesta: Es un conjunto de sentencias agrupadas. Se pueden agrupar varias sentencias ya sean simples u otras sentencias compuestas en una con el uso de delimitadores (Ada y Pascal: Begin/End; C, C++ y Java: { }) dentro de un bloque.

Ejercicio 2: Analice como C implementa la asignación.

La asignación en C se realiza mediante el operador = (igual). La sintaxis básica es: variable = expresión;

variable: Es un identificador que representa una ubicación en memoria donde se almacenará el valor.

expresión: Puede ser una constante, otra variable, una operación aritmética, una llamada a función, etc., cuyo resultado se evalúa a un valor compatible con el tipo de la variable.

- Las sentencias de asignación devuelven valores. Se define la sentencia de asignación como una "expresión con efectos colaterales".
- En C se evalúa la asignación de derecha a izquierda (a=b=c=0 → 0 se asigna a "c", "c" a "b" y "b" a "a").
- Si se usa asignaciones en condiciones primero asigna y luego evalúa, además, toma como verdadero todo resultado distinto que 0.

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 puede producir efectos secundarios que afecten el resultado final, dependiendo del orden de evaluación de las subexpresiones. Las asignaciones no solo almacenan valores, sino que también pueden modificar variables que se usan en otras partes de la misma expresión.

1. **Modificación de una variable usada en la misma expresión**:

c ··· Copy

```c
int x = 5;
int y = (x = 10) + x;
```

- Aquí, `x = 10` asigna 10 a `x` y devuelve 10 (el valor asignado).
- Luego, se suma `x`, que ahora vale 10.
- Resultado: `y = 10 + 10 = 20`.
- **Efecto secundario**: La asignación `x = 10` cambia el valor de `x`, afectando el término `x` en la expresión.

3. **Uso de operadores de incremento con asignación**:

```c
int x = 5;
int y = x++ + (x = 10);
```

- `x++` usa el valor actual de `x` (5) y luego incrementa `x` a 6.
- Pero `(x = 10)` asigna 10 a `x`, sobrescribiendo el incremento.
- El orden de evaluación entre `x++` y `x = 10` no está definido, lo que puede dar resultados como:
  - `y = 5 + 10 = 15` (si `x++` se evalúa primero).
  - O un resultado impredecible, ya que modificar `x` múltiples veces sin un punto de secuencia es **comportamiento indefinido**.

4. **Asignación encadenada con efectos secundarios**:

```c
int a = 1, b = 2;
a = b = a + 1;
```

- Primero, se evalúa `a + 1` (1 + 1 = 2).
- Luego, `b = 2` asigna 2 a `b`.
- Finalmente, `a = 2` asigna 2 a `a`.
- Resultado: `a = 2`, `b = 2`.
- **Efecto secundario**: La asignación a `b` depende del valor modificado de `a + 1`, y ambas variables se alteran en la misma expresión.

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.

La terminología "circuito corto" y "circuito largo" se refiere a cómo se evalúan las expresiones lógicas en ciertos contextos. Son técnicas utilizadas en la evaluación de expresiones booleanas que involucran operadores lógicos tipo AND y OR (no aplica a XOR).

Circuito corto: Significa que los operandos de una expresión lógica se evalúan de izquierda a derecha, pero solo hasta encontrar el primero que determina el resultado final. En ese punto, la evaluación se detiene y no se evalúan los operandos restantes.

- La conjunción (“y”/”and”) da como resultado verdadero únicamente cuando ambos términos son verdaderos. Si el primer término es falso, no es necesario evaluar el segundo: el resultado será falso.
- La disyunción (“o”/”or”) da como resultado falso únicamente cuando ambos términos son falsos. Si el primer término es verdadero, no es necesario evaluar el segundo: el resultado será verdadero.

Circuito largo: Significa que todos los operandos se evalúan, sin importar si el primero ya determina el resultado final.

```
function buscarEnVector(v: Tvector; n: integer): boolean;
var
    i: integer;
begin
    i := 1;
    while (i ≤ DIMF) and (v[i] <> n) do
        i := i + 1;
    buscarEnVector := (i ≤ DIMF); // Retorna true si se encontró el numero, false si no
end;
```

Este código de Pascal gracias al circuito corto no da error, se trata de una búsqueda desordenada en un vector de enteros, dónde se verifica que el índice (i) no supere el número de elementos que contiene el arreglo. Cuando se haya superado esa cantidad, la segunda condición no se evalúa. En el caso de usar circuito largo, se intentaría acceder a una posición fuera de rango del arreglo, dando error.

Ejercicio 5: ¿Qué regla define Delphi, Ada y C para la asociación del else con el if correspondiente? ¿Cómo lo maneja Python?

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

Como regla se utiliza la coincidencia de las palabras clave begin y end para determinar la asociación del else con el if correspondiente más cercano. El else se asocia con el if que tiene la misma palabra clave begin y end más cercana y que no tiene un else asociado.

Python

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

Ejercicio 6: ¿Cuál es la construcción para expresar múltiples selecciones que implementa C? ¿Trabaja de la misma manera que la de Pascal, ADA o Python?

En C:

- Constructor Switch seguido de (expresión).
- Cada rama Case es "etiquetada" por uno o más valores constantes (enteros o char).
- Si coincide con una etiqueta del Switch se ejecutan las sentencias asociadas, y se continúa con las sentencias de las otras entradas. (chequea todas salvo que exista un break).
- Existe la sentencia break, que provoca la salida de cada rama (sino continúa), es opcional.
- Existe opción default que sirve para los casos que el valor no coincida con ninguna de las opciones establecidas, es opcional.
- Si un valor no cae dentro de alguno de los casos de la lista de casos y no existe un default no se provocará un error por esta acción, pero se podrían generar efectos colaterales dentro del código.
- El orden en que aparecen las ramas no tiene importancia.

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.

En Ada:

- Constructor: Case expresión is con when y end case.
- 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.
- 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.

En Python:

- Constructor: match expresión: seguido de bloques case patrón: indentados.
- Tipos de expresiones: La expresión puede ser de cualquier tipo (enteros, cadenas, listas, objetos, etc.), y los patrones pueden ser literales, variables, estructuras de datos o patrones complejos.
- Cobertura de casos: No es obligatorio contemplar todos los valores posibles de la expresión. Si no hay coincidencia y no se usa _, no hay error, pero no se ejecuta nada.
- Indicador de acción: case patrón: define el patrón a comparar, seguido de un bloque indentado con las sentencias a ejecutar si el patrón coincide.
- Cierre: el fin de la estructura se indica por la desindentación del código.
- Caso por defecto: case _: (opcional) actúa como el caso por defecto, capturando cualquier valor que no coincida con los patrones anteriores.
- Posición de case _: Debe ser el último case, ya que cualquier case posterior sería inalcanzable (aunque Python no genera error si se coloca antes, simplemente no se ejecutará).
- Ejecución exclusiva: Una vez que un case coincide, se ejecuta su bloque y el match termina; no se evalúan otras ramas.

Sintaxis:

```c
switch (expresión) {
    case valor1:
        // Código
        break;
    case valor2:
        // Código
        break;
    default:
        // Código si no coincide
}
```

```pascal
case expresión of
    valor1:
        begin
            // Código
        end;
    valor2:
        begin
            // Código
        end;
    else
        // Código si no coincide
end;
```

```ada
case expresión is
    when valor1 =>
        -- Código
    when valor2 =>
        -- Código
    when others =>
        -- Código si no coincide
end case;
```

```python
match expresión:
    case valor1:
        # Código
    case valor2:
        # Código
    case _:
        # Código si no coincide
```

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.

Este código podría genera error en Pascal estándar ya que este "no permite" que se modifiquen los valores del límite inferior, límite superior, ni del valor de la variable de control, y dentro del bucle (por el procedimiento A) se viola esta regla, al modificar el índice utilizado en el for. Aún así, la modificación se realiza afuera de la misma unidad del for y no genera error.

En Ada el código no compilaría debido a la prohibición de modificar la variable de control i dentro del bucle for.

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

En mi versión de Pascal el bucle se ejecuta. Pero en Ada, el compilador genera un error.

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.

El código en Pascal no cubre todos los valores posibles de puntos (un entero puede tomar valores como 0, 6, 7, 8, 9, 11, etc.). En Ada, esto provocaría un error de compilación porque no hay un when others ni se especifican todos los valores posibles.

ada

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   puntos : Integer;
begin
   puntos := 10;  -- Ejemplo de valor
   case puntos is
      when 1 .. 5 =>
         Put_Line("No puede continuar");
      when 10 =>
         Put_Line("Trabajo terminado");
      when others =>
         Put_Line("Valor no especificado");
   end case;
end Main;
```

Sintaxis: case puntos is y when valor => son la forma correcta en Ada.

Cobertura: Se agrega when others para cubrir todos los valores de Integer no especificados (e.g., 0, 6, 7, 8, 9, 11, etc.), evitando el error de compilación.

Salida: Si puntos = 10, imprime "Trabajo terminado". Si puntos = 3, imprime "No puede continuar". Si puntos = 7, imprime "Valor no especificado".

En C no se requiere cubrir todos los casos. Si no hay coincidencia y no hay default, simplemente no se ejecuta ninguna rama. El rango 1..5 no es directamente compatible con switch-case en C. Usar case 1..5: generaría un error de sintaxis. En C, cada case debe ser un valor constante individual, no un rango.

c

```c
#include <stdio.h>

int main() {
    int puntos = 10;  // Ejemplo de valor
    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:
            printf("Valor no especificado\n");
    }
    return 0;
}
```

Rangos en switch: Se lista cada valor (case 1: a case 5:) para emular el rango 1..5. Sin break entre ellos, se ejecuta el mismo bloque para todos (efecto "fall-through").

Cobertura: En C, no es obligatorio cubrir todos los casos, pero se agregó un default (o else) para manejar valores no especificados, como buena práctica.

Salida: Si puntos = 10, imprime "Trabajo terminado". Si puntos = 3, imprime "No puede continuar". Si puntos = 7, imprime "Valor no especificado".

Ejercicio 9: Qué diferencia existe entre el generador YIELD de Python y el return de una función. De un ejemplo donde sería útil utilizarlo.

Comportamiento:

- return: Termina la ejecución de la función inmediatamente y devuelve un único valor (o nada si no se especifica). La función no se puede reanudar después de un return.
- yield: Convierte la función en un generador, que produce una secuencia de valores uno a la vez. La ejecución se pausa en cada yield, permitiendo reanudarla más tarde, lo que ahorra memoria al generar valores bajo demanda.

Estado:

- return: No conserva el estado entre llamadas; cada invocación reinicia la función desde el principio.
- yield: Mantiene el estado interno del generador entre iteraciones, retomando desde donde se dejó.

Uso de memoria:

- return: Devuelve todos los datos de una vez, lo que puede consumir mucha memoria si el resultado es grande.
- yield: Genera valores de forma iterativa, ideal para manejar secuencias grandes sin almacenarlas completas en memoria.

Tipo de resultado:

- return: Devuelve un valor o estructura de datos única (e.g., lista, entero).
- yield: Devuelve un objeto generador que se puede iterar con un bucle o next().

Un caso práctico donde yield es útil es al generar una secuencia infinita o procesar datos grandes, como números de la serie de Fibonacci. Esto evita crear una lista completa en memoria. En caso de usar return si n es muy grande (e.g., 1,000,000), la lista fib ocupará mucha memoria.

Ejercicio 10: Describa brevemente la instrucción map en javascript y sus alternativas.

El método map en JavaScript es una función de los arrays que crea un nuevo array aplicando una función callback a cada elemento del array original. No modifica el array original y devuelve un nuevo array con los resultados. Sintaxis: array.map(callback(elemento, índice, array) [, thisArg]).

Alternativas

<u>forEach:</u>

- Itera sobre los elementos y ejecuta una función por cada uno, pero no devuelve un nuevo array (solo realiza efectos secundarios).
- Ejemplo: numeros.forEach(num => resultado.push(num * 2));

for:

- Permite modificar el array original o crear uno nuevo.
- for (let i = 0; i < numeros.length; i++) { resultado.push(numeros[i] * 2); }

<u>for...of:</u>

- Itera sobre los elementos de un array de forma más legible que un for tradicional, pero requiere crear manualmente un nuevo array.
- for (const num of numeros) { resultado.push(num * 2); }

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.

Java implementa instrucciones para el manejo de espacios de nombres a través de paquetes (packages). Los paquetes son el mecanismo principal para organizar y gestionar nombres en Java, evitando colisiones y estructurando el código.

Un *espacio de nombres (namespace)* es un mecanismo que permite organizar código agrupando identificadores (nombres de clases, funciones, variables, etc.) bajo un nombre único, evitando conflictos entre nombres idénticos en diferentes partes de un programa o bibliotecas.

En Java, los espacios de nombres se implementan mediante **paquetes**:

- **Declaración**: Se define un paquete al inicio de un archivo con la instrucción `package`.

```java
package com.ejemplo.mipaquete;
public class MiClase {
    // Código
}
```

- **Importación**: Para usar clases de otro paquete, se usa `import`.

```java
import com.ejemplo.mipaquete.MiClase;
// O importar todo el paquete
import com.ejemplo.mipaquete.*;
```

PHP implementa espacios de nombres desde la versión 5.3 con la instrucción namespace.

- **Declaración**:

```php
namespace MiProyecto\SubEspacio;
class MiClase {}
```

- **Uso**:
  - Importar con `use`:

```php
use MiProyecto\SubEspacio\MiClase;
$obj = new MiClase();
```

Ámbitos globales y locales: \ al inicio indica el espacio global (e.g., \strlen()).

Alias: use MiProyecto\SubEspacio\MiClase as Alias; permite renombrar para evitar conflictos.

Soporte dinámico: Los nombres pueden resolverse en tiempo de ejecución.

Evita colisiones: Útil en proyectos grandes o al usar bibliotecas de terceros (e.g., Composer).

Python no tiene una instrucción explícita para espacios de nombres como namespace, pero los implementa mediante módulos y paquetes.

- **Declaración**:
  - Un módulo es un archivo `.py` (e.g., `modulo.py`).
  - Un paquete es un directorio con un archivo `__init__.py`.
  - Ejemplo de estructura:

```text
mi_paquete/
    __init__.py
    modulo.py
```

- **Uso**:

```python
import mi_paquete.modulo
from mi_paquete.modulo import MiClase
```

Espacios implícitos: Cada módulo crea su propio espacio de nombres.

Importaciones relativas: from .modulo import MiClase dentro de un paquete.

Evita colisiones: mi_paquete.modulo.MiClase y otro_paquete.modulo.MiClase pueden coexistir.

Flexibilidad: Los módulos son objetos dinámicos, permitiendo inspección y manipulación en tiempo de ejecución.