Conceptos y Paradigmas de lenguajes de Programación
2024

Práctica Nro. 8
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?

Ejercicio 2: Analice como C implementa la asignación.

Ejercicio 3: ¿Una expresión de asignación puede producir efectos laterales que afecten al resultado final, dependiendo de cómo se evalúe? De ejemplos.

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.

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

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

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

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

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.

Ejercicio 1

- **Sentencia**
  - 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.
- **Sentencia Compuesta**
  - Es un conjunto de sentencias agrupadas que se ejecutan secuencialmente. En muchos lenguajes de programación, las sentencias compuestas se encierran entre llaves ({}) 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

- **Sentencia de Asignación en C**
  - Las Sentencias de asignación devuelven valores.
  - 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

- Si, una asignación puede producir efectos colaterales que afecten al resultado final dependiendo de cómo se evalúe. Un ejemplo de esto es el previamente mencionado de una asignación dentro de una condición.

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.
    - **Operador(“y” /”and” / “&&”)**
        - En un circuito largo, 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” / “||”)**
        - En un circuito largo, ambas partes de la expresión se evalúan independientemente del valor de la primera parte. Esto asegura que se realicen todas las comprobaciones necesarias.
- **Ejemplo que no da error con Circuito Corto pero si con Circuito Largo**

- ```c
#include <stdio.h>

int main() {
    int a = 0;
    int b = 10;
    int result;

    // Evaluación de circuito corto con '&&' (AND)
    if (a != 0 && (b / a) > 1) {
        result = b / a;
    } else {
        result = -1; // Evita la división por cero
    }

    printf("Resultado con evaluación de circuito corto: %d\n", result);

    return 0;
}
```

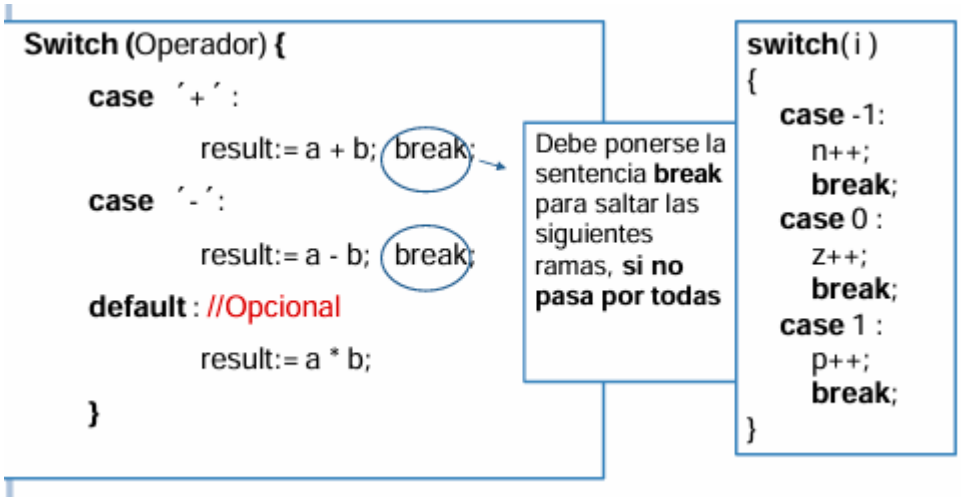
# Ejercicio 5

- **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**
  - 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

- **Selección Múltiple 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)
- 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, por eso es importante chequear el valor si puede tomar valores fuera del rango de la lista de casos.
- El orden en que aparecen las ramas no tiene importancia.



- **Selección Múltiple en Pascal**
  - Usa palabra reservada case seguida de una variable de tipo ordinal o una expresión y la palabra reservada of.
  - La variable-expresión a evaluar es llamada “selector".
  - Lista Las sentencias de acuerdo con diferentes valores que puede adoptar la variable (los “casos”). Llevan etiquetas.
  - No importa el orden en que aparecen los casos.
  - Puede Existir un bloque else para el caso que la variable adopte un valor que no coincida con ninguna de las sentencias de la lista de casos.
  - Para finalizar se coloca un “end;” (no se corresponde con ningún “begin” que exista).
  - Es inseguro si no se establece un bloque else y sucede que la variable toma un valor que no está dentro de la lista de casos.

El formato es el siguiente:

```
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)

- **Selección Múltiple en Ada**
  - Las expresiones pueden ser solamente de tipo entero o enumerativas.
  - En Las Elecciones del case se deben estipular "todos" los valores posibles que puede tomar la expresión.
  - El when se acompaña con => para indicar la acción a ejecutar si se cumple la condición.
  - Tiene la cláusula Others que se puede utilizar para representar a aquellos valores que no se especificaron explícitamente.
  - Others"debe" ser la última opción antes del end;
  - Después que una rama es ejecutada el Case entero finaliza. (no pasa por otras ramas)
  - No pasa la compilación si
    - NO se coloca la rama para un posible valor y/o NO aparece la opción Others en esos casos.

Ejemplo 1

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

Importante para el programador:

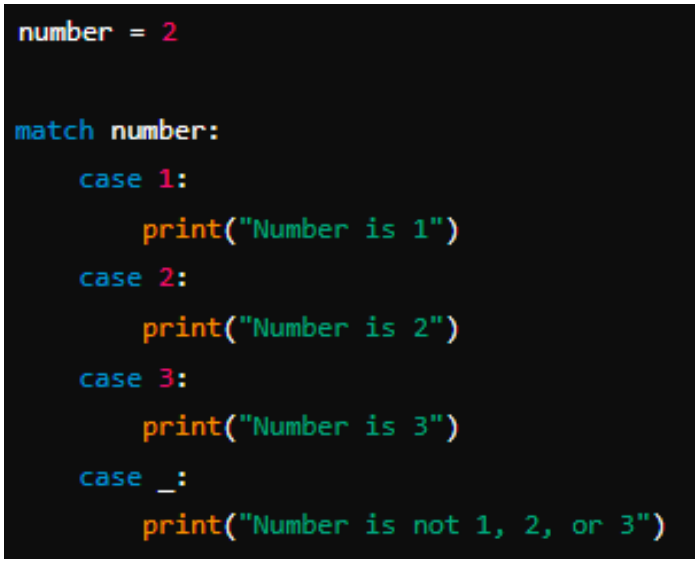
La cláusula **others** se **debe colocar** porque las etiquetas de las ramas NO abarcan todos los posibles valores de Operador

Debe ser la última

Ejemplo 2

```
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. (LUN)
end case;
```

- **Sentencia 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.
- 

```
number = 2

match number:
    case 1:
        print("Number is 1")
    case 2:
        print("Number is 2")
    case 3:
        print("Number is 3")
    case _:
        print("Number is not 1, 2, or 3")
```

Ejercicio 7

Inciso a

- 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’”.
- 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.

Inciso b

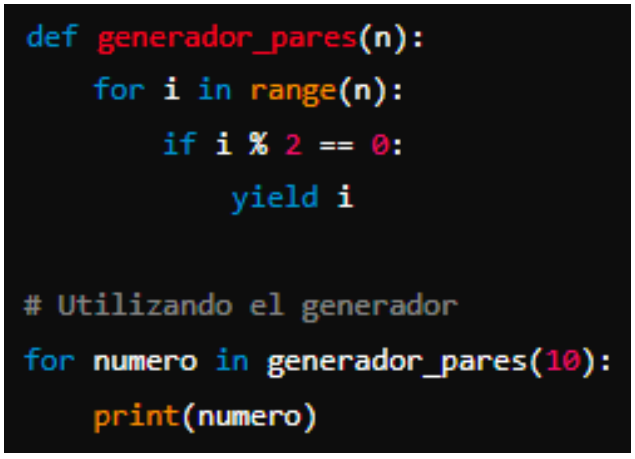
- Con Ada no trabajé (por suerte) así que no puedo responder pero con Pascal no creo que en versiones específicas genere error, solo efecto colateral.

Ejercicio 8

- 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.

- 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

- **YIELD**
  - Yield se utiliza en generadores para producir una serie de valores en lugar de uno solo.
  - Cuando se encuentra una declaración yield, la función se detiene temporalmente y devuelve el valor especificado. Sin embargo, la función no se termina por completo; en su lugar, mantiene su estado actual y puede continuar ejecutándose cuando se solicita el próximo valor.
  - El generador recuerda su estado y puede reanudar la ejecución desde el punto donde se detuvo anteriormente.
- **RETURN**
  - Return se utiliza para devolver un valor desde una función y terminar la ejecución de la función en ese punto.
  - Cuando se encuentra una declaración return, la función se detiene y devuelve el valor especificado, si lo hay, al lugar donde fue llamada.
  - Después de que se ejecuta una declaración return, cualquier código que siga a esa declaración dentro de la función no se ejecutará.
- **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 con 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.
- **Alternativas**
    - **ForEach**
        - Itera sobre cada elemento de un array y ejecuta una función para cada elemento, pero no devuelve un nuevo array, sino que se utiliza principalmente para realizar operaciones con efectos secundarios en cada elemento.
    - **Filter**
        - Crea un nuevo array con todos los elementos que pasan cierta condición proporcionada en una función de prueba.
    - **Reduce**
        - Reduce un array a un solo valor aplicando una función a cada elemento y acumulando el resultado.
    - **FlatMap**
        - Similar a 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.