Trabajo práctico de Conceptos y Paradigmas de Lenguajes de Programación

Grupo 37

Lenguajes: C y Python

A. Escriba 2 fragmentos de código de mínimo 5 y máximo 10 líneas de extensión que permitan ejemplificar al menos dos criterios de evaluación en los lenguajes asignados. Puede citar un ejemplo para cada criterio elegido y compararlo entre los lenguajes, colocando una breve descripción textual para cada uno.

Python	C

try:
print(1/0)
except ZeroDivisionError as e:
print("no se puede dividir entre 0!")
list = [1,2,3,4,5]
	#include <stdio.h>
	struct Nodo{ //declaración de la estructura de lista enlazada
	  int dato;
	  struct Nodo *siguiente;
	};
	int main(){
	  int divisor = 0;
	  if(divisor == 0){
	  printf("No se puede dividir por cero\n");
	  }
	}

- Abstracción: En los fragmentos de código se puede observar la declaración de una lista tanto en C como en Python. En Python con simplemente ingresar los elementos que se van a agregar en la lista entre corchetes([]) y separarlos con comas(,), el lenguaje interpreta que estamos queriendo hacer una lista y se encarga de implementar la estructura.

  En cambio, para hacer una lista en C, tenemos que ocuparnos nosotros como programadores de expresar la estructura de la lista y la manejamos mediante punteros. A su vez, el manejo de la lista queda también del lado del programador. Por ejemplo, para agregar un elemento al final de la lista debemos desplazarnos mediante los punteros hasta el último elemento y agregarlo en esa posición, mientras que en Python podemos hacerlo simplemente con lista.append().

- Simplicidad y legibilidad: La legibilidad del código en Python contrasta con la complejidad de C. En Python, la declaración de listas es simple y legible, con una sintaxis clara y concisa. En cambio, en C, la construcción de listas requiere el uso de palabras reservadas y una estructura más robusta, lo que puede resultar menos legible y más propenso a errores. En resumen, Python ofrece una sintaxis más amigable y comprensible, mientras que en C, la complejidad del lenguaje puede dificultar la legibilidad del código.

- Confiabilidad: Python tiene un manejo de excepciones que viene incorporado con el lenguaje, mientras que en C debemos ocuparnos nosotros de implementar algo que

haga algo similar. Por otro lado, otra cosa que los hace diferentes en términos de confiabilidad es que C hace un chequeo de tipos en compilación, mientras que en Python se hace en ejecución.

B. Cite 2 ejemplos de código (mínimo 5, máximo 15 líneas) para cada lenguaje asignado que permitan analizar A NIVEL SINTÁCTICO 2 estructuras de control de manera comparada entre los lenguajes. Coloque una breve descripción a cada ejemplo.

Python	C

`input= "s"`
`if usuario == "S" or usuario == "s":`
`  print("El usuario eligió S")`
`  print("Verdadero")`
`elif usuario == "N" or usuario == "n":`
`  print("Falso")`
`else:`
`  print("Valor incorrecto")`
	`#include <stdio.h>`
	`int main() {`
	`  char input = 's'`
	`  if (input == 's' || input == 'S'){`
	`  printf("El usuario eligió S\n");`
	`  printf("Verdadero\n");`
	`  }`
	`  else if (input == 'n' || input == 'N')`
	`  printf("Falso\n");`
	`  else`
	`  printf("Valor Incorrecto\n");`
	`}`

- La sintaxis abstracta en ambos lenguajes es similar, se introduce la palabra reservada “if”, la condición y el símbolo que da inicio al bloque a ejecutar en caso de que la condición sea verdadera. En términos de sintaxis concreta, sí que son bastante diferentes. En primer lugar, el bloque a ejecutar en C se encierra entre llaves (cuando tiene más de una línea) mientras que en Python se indenta. Por otro lado, en Python, la estructura “elif” se utiliza para lo que en C se conoce como “else if”, lo cual ofrece una sintaxis más clara y concisa. En tanto a las diferencias de sintaxis pragmática, es más legible y claro el uso del “or” de Python que el “||” de C para definir la suma lógica.

Python	C

`x = 10`
`while x > 0:`
`  print(x)`
`  x-=1`
	`#include <stdio.h>`
	`int main(){`
	`  int x = 10;`
	`  while (x > 0) {`
	`  printf("%d\n", x);`
	`  x--;`
	`  }`
	`}`

- Se puede apreciar una sintaxis abstracta similar en ambos lenguajes. Se inicializa el bloque con la palabra reservada “while”, seguida por la condición de corte y luego el bloque. No obstante, tienen diferencias a nivel concreto. C encierra los bloques entre corchetes mientras que en Python simplemente se le da comienzo al bloque con “:” y se usa la indentación para determinar qué líneas abarca. A su vez, las sentencias de C se finalizan con “;” mientras que Python no usa ningún carácter con ese fin. En términos de sintaxis pragmática, C no incluye por defecto una forma de mostrar por pantalla ni leer del teclado, por lo que se debe incluir la línea “#include <stdio.h>” con ese fin, mientras que en Python dicha funcionalidad viene por defecto.

C. Cite un ejemplo de código (5 líneas mínimo, 10 líneas máximo) para cada lenguaje asignado donde pueda distinguirse con claridad chequeos de semántica estática y dinámica a la hora de ejecutar/interpretar código. Debe poder entenderse con claridad, más allá de la naturaleza del lenguaje, distintos tipos de errores que se controlan y el momento en que dichos chequeos se realizan hasta poder ejecutar propiamente el programa.

C	Python

`int main(){`
`  int a = 1;`
`  a = "hola";// error semántica estática`
`  int arr[2];`
`  arr[0] = 1;`
`  arr[1] = 2;`
`  print(arr[2]);// error semántica dinámica`
`}`
	`texto = "Hello World"`
	`numero = int(texto)`
	`print(numero)`
	`#Intento de llamar  a una función que no existe`
	`print(funcion_inexistente())`

- En el ejemplo en C, se está intentando asignar una string a una variable previamente declarada como int. Este error es de semántica estática y será detectado por el compilador. Por otro lado, en el fragmento de código se puede observar la creación de un array de 2 elementos y el posterior intento de acceder al elemento en la posición 2 (que está fuera de su rango). Este es un error que será detectado al hacer el chequeo de semántica dinámica.

- En Python, los errores de semántica estática no ocurren porque Python es un lenguaje de tipado dinámico. Esto significa que las comprobaciones de tipo y otros análisis que suelen ser estáticos no se realizan hasta que se ejecuta el código. Por otro lado, en el ejemplo brindado se pueden observar dos errores detectados en el chequeo de semántica dinámica. Por un lado, en la línea 2 se está intentando hacer un casteo que

no es válido (de string a int). Por el otro, en la 5 se llama a una función que no está declarada.

D. Defina un ejemplo de código (5 líneas mínimo,15 líneas máximo) en cada lenguaje seleccionado donde se visualicen los distintos tipos de variables que soporta el lenguaje en cuanto al tiempo de vida, y realice una tabla donde se sitúen todos los identificadores junto con los valores de sus atributos tal como se vió en la práctica correspondiente.

Ejemplo de C:

```c
#include <stdio.h>
int variable_global = 10;  // Variable global
int *puntero;
void mi_funcion() {
    static int variable_estatica;  // Variable estática local
    int variable_automatica = 20;  // Variable automática local
    int variable_global = 5;
    puntero = &variable_global;  // Puntero que apunta a la variable global.
}
int main() {
    mi_funcion();
    printf("Variable global fuera de la función: %d\n", variable_global);
}
```

Identificador	L-Valor	R-Valor	Alcance	Tiempo de vida
variable_global(2)	automática	10	3-7 10-13	2-13
mi_funcion	-	-	5-13	4-9
Variable_estática	estática	0	6-9	<1-13>
variable_automatica	automática	basura	7-9	4-9
puntero	automática	null	4-13	2-13
*puntero	dinámica	basura	4-13	8-9
variable_global(7)	automática	5	8-9	4-9
main	-	-	11-13	10-13

Ejemplo en Python:

```
1. x = 11 // Variable global
2. def main():
3.   x = 1 // Variable local
4.   y = 5
5.   print(x)
6. def mi_function():
7.   global x
8.   x = 7
```

Identificador	L-Valor	R-Valor	Alcance	Tiempo de vida
x(1)	dinámica	11	2-3 6-8	1-8
main	-	-	3-8	2-6
x(3)	dinámica	1	4-5	3-6
y	dinámica	5	5-6	4-6
mi_function	-	-	7-8	6-8
x(8)	dinámica	7	8-8	6-8

Segunda parte

A. Cite un ejemplo de código para cada lenguaje seleccionado donde se aprecien los distintos tipos de parámetros soportados y su uso. Cada fragmento puede tener una breve explicación textual. En la misma debe quedar claro el modo de ligadura que utiliza el lenguaje (posicional, por nombre, etc), si requieren o no ser tipados, o cualquier otra característica que considere relevante.

En C están los parámetro de tipo IN por valor, IN por valor constante y por último IN-OUT por referencia y OUT por resultado de función. El modo de ligadura es posicional, es decir que los parámetros se pasan por posición, y su tipado es estático, lo que significa que los tipos de los parámetros se deben conocer en el momento de compilación.

```c
#include <stdio.h>
void inPorValor(int a);
void inOutPorReferencia(int *b);
void inValorConstante(const int c);
int outResultadoFuncion(int w);

int main() {
    int x = 10;
    int y = 20;
    int z = 30;
    int w = 40;


    inPorValor(x);
    printf("Después de inPorValor, x = %d", x); //imprime 10

    inOutPorReferencia(&y);
    printf("Después de inOutPorReferencia, y = %d", y); // imprime 200

    inValorConstante(z);
    printf("Después de inValorConstante, z = %d", z); // imprime 30

    printf("Antes de outResultadoFuncion, w = %d", w); // imprime 40
     w = outResultadoFuncion(w); // se cambia el valor de w por el valor de
    la función
    printf("Después de outResultadoFuncion, w = %d", w); // imprime 1600


    return 0;
```

```c
}

void inPorValor(int a) {
    a = 100; // Este cambio no afecta a la variable original
    printf("Dentro de inPorValor, a = %d", a); // imprime 100
}

void inOutPorReferencia(int *b) {
    *b = 200; // Este cambio afecta a la variable original
    printf("Dentro de inOutPorReferencia, *b = %d", *b); // imprime 200
}

void inValorConstante(const int c) {
    // c = 300; // Esto causaría un error de compilación
    printf("Dentro de inValorConstante, c = %d", c); // imprime 30
}

int outResultadoFuncion(int w){
  return w * w
}
```

En Python se envían objetos que pueden ser “inmutables” o “mutables”. Si un objeto es inmutable, se maneja por su valor asignado, mientras que si es mutable (como las listas), no se duplica para su manipulación, sino que se modifica directamente. En resumen, los objetos inmutables se comportan como valores, mientras que los mutables se modifican en su lugar. La ligadura en Python se realiza en ejecución y se trata de un lenguaje fuertemente tipado.

```python
def sumar(x, y):
  """Suma dos números y devuelve el resultado."""
  return x + y

resultado = sumar(y=5, x=3)  # Ligadura por nombre
print(resultado)  # Salida: 8

def modificar_lista(lista):
  """Cambia un elemento a la lista."""
  lista[0] = 0

lista = [1, 2, 3]
modificar_lista(lista)
print(lista)  # Salida: [0, 2, 3]
```

B. Ejemplifique la fortaleza del sistema de tipos de ambos lenguajes con al menos dos ejemplos de código (5 líneas mínimo, 10 líneas máximo) que sustenten la definición otorgada.

C es débilmente tipado lo cual ofrece una gran flexibilidad. Esto se visualiza, por ejemplo, con el hecho de que una variable de tipo int pueda ser sumada a un caracter sin la necesidad de hacer una conversión explícita (realizada por el compilador). En cambio, Python es fuertemente tipado. Esto hace que posea reglas más estrictas sobre las operaciones que se pueden realizar entre sus tipos de datos, por ejemplo, para realizar el mismo tipo de conversión que ejemplificamos para C, habría que especificarle que queremos hacer la conversión de tipos. En caso contrario esto nos dará un error debido a la restricciones que este lenguaje posee

Ejemplo en C:

```c
#include <stdio.h>
int main() {
    int entero = 10;
    char caracter = '2';
    // Asignación de un entero a un float
    printf("Valor: %d\n", entero + caracter);//imprime 60 (ascii del 2+10)
    return 0;
}
```

Ejemplo en Python:

```python
entero = 10
carácter = '2'

# Asignación de un entero a un float en Python
concatenar = entero + caracter # Esto generará un TypeError
concatenar = entero + int(caracter) # Esto si funciona y no da error
print("Valor:", concatenar)
```

C. Cite el paradigma de manejo de excepciones correspondiente a cada lenguaje asignado, y coloque un ejemplo para cada uno que describa claramente el circuito posible al alcanzar una excepción y que considere todas las variantes de manejo.

Python maneja las excepciones bajo el modelo de terminación, es decir, una vez que se produce la excepción, esta es manejada pero no retorna al punto en donde se produjo, sino continua la ejecución a partir de la finalización del manejador.
Utiliza las sentencias try - except - else - finally

```python
def division(a, b):
    try:
        resultado = a / b
    except ZeroDivisionError as e:
        print("Error en la división: {e}")
    else:
        return result
    finally:
        print("Operación de división completa.")
```

El código de Python tiene el siguiente funcionamiento:
Primero se ejecuta el bloque try, de no levantarse ninguna excepción el bloque except no se ejecutaría, se completa la ejecución del try y se ejecuta el bloque else, en cualquier caso, se levante o no una excepción el bloque finally será ejecutado siempre.

En caso de ocurrir una excepción en el bloque try este bloque es finalizado y se ejecutará su manejador, en caso de no encontrarse un manejador para la excepción, este se buscará estáticamente en declaraciones try por fuera del ya definido. Luego se realiza una búsqueda dinámica, en donde verificá si existe algún manejador para la excepción de aquellos que hayan invocado al módulo (en este caso, la función division) En caso de no encontrar un manejador, la ejecución frena con un mensaje de error.

Para lanzar excepciones explícitamente se utiliza la palabra “raise”. Podríamos en el código provisto verificar si el divisor es 0 y en ese caso hacer raise ZeroDivisionErrror, pero esta excepción ya está provista por el lenguaje.

C por el contrario no tiene manera de manejar excepciones nativas al lenguaje, esto se hace a través de la verificación de errores después de las operaciones y el manejo de los mismos a través de estructuras de control como if’s y switch.

```c
#include <stdio.h>
int divide(int a, int b) {
    if (b == 0) {
        return -1;
    }
    return a / b;
}

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

```
    int result;

    result = divide(a, b);
    if (result == -1) {
        printf("Error en la división");
    }

    printf("Resultado: %d\n", result);
}
```

En el código provisto de C, la función divide se encarga de verificar si la división por 0 es posible, de no ser posible retorna -1, esto luego es comprobado por la función main para ver si se generó error o no.

D. Conclusión sobre el trabajo: Realice una conclusión de 20 líneas como máximo, mencionando los aportes que le generó la realización del trabajo en comparación con sus conocimientos previos de los lenguajes asignados.

La realización de este trabajo práctico nos ha permitido aprender más sobre los lenguajes de programación elegidos profundizando aún más los conceptos explicados en clase. Pudimos aprender sus características fundamentales y cómo se aplican en contextos prácticos.

Explorar las diferencias en el manejo de variables entre C y Python nos permitió apreciar cómo la gestión automática de memoria en Python simplifica el desarrollo y reduce el riesgo de errores relacionados con la gestión de memoria manual permitiéndonos escribir programas más eficientes y menos propensos a errores.

Por otro lado, el estudio de los sistemas de tipos en C y Python nos dio herramientas para diseñar programas más seguros. Aprendimos que en C, aunque el tipado débil ofrece flexibilidad, requiere tener más cuidado sobre los tipos de datos para evitar comportamientos inesperados. Por el contrario, en Python, el tipado fuerte nos asegura que los tipos de datos sean coherentes y previsibles, lo cual nos asegura que se mantenga la integridad de datos.

Por último, aprendimos que en Python el manejo de excepciones nos permite manejar errores de manera eficaz, mejorando la resistencia del código frente a situaciones imprevistas. Por el contrario, el manejo de errores en C es más manual. Esto nos ha enseñado lo importante que es un diseño cuidadoso y una planificación anticipada para manejar eficazmente los errores.

En resumen, este trabajo práctico nos ha proporcionado no solo más conocimientos técnicos, sino también una nueva perspectiva sobre cómo elegir y aplicar el lenguaje de programación más adecuado para cada proyecto. Mejoramos nuestra habilidad de escribir código eficiente, seguro y robusto, adaptado a los desafíos y requisitos específicos de cada situación.