Práctica 3

1) La **semántica** de un lenguaje de programación define el **significado** de sus construcciones y expresiones. Es decir, describe cómo se comporta un programa cuando se ejecuta y qué efectos produce en términos de cambios en el estado del sistema, cálculos realizados y resultados obtenidos.

2) Compilar un programa significa traducir el código fuente escrito en un lenguaje de programación de alto nivel (como C, Java o Python) a un código de bajo nivel (como lenguaje ensamblador o código máquina) que pueda ser ejecutado por la computadora. Este proceso es realizado por un **compilador**.
   Pasos:
   - **Análisis Léxico:** divide el programa en elementos y ve si son tokens válidos.
   - **Análisis Sintáctico:** se identifican estructuras ayudándose de los tokens.
   - **Análisis Semántico (Semántica Estática):** realiza comprobación de tipos, nombres, variables, etc.
   - **Generación de Código Intermedio:** genera una representación independiente de la máquina.
   - **Optimización de Código Intermedio**
   - **Generación de Código Objeto:** construye el programa ejecutable. Se traducen módulos que luego se enlazan.
   - **Optimización de Código Objeto:** mejora velocidad, tamaño, elimina código muerto, etc.

   La semántica interviene principalmente en el **análisis semántico**.
   - Su importancia radica en garantizar que el código **tenga sentido y sea válido** antes de la generación del código final.
   - Sin esta verificación, un programa podría compilarse sin errores sintácticos pero **no funcionar correctamente** o producir resultados incorrectos.

3) No es lo mismo. La interpretación se realiza en la ejecución, mientras que la compilación se hace antes. En el caso del primero solo se analizan las acciones ejecutadas (orden lógico), mientras que en la compilación se verifican todas las líneas de manera secuencial (orden físico). La interpretación es más lenta de ejecutar y ocupa menos espacio, mientras que la compilación es más rápida pero ocupa mucho espacio. En cuanto a la detección de errores, la interpretación te permite ver dónde se encuentra el error, mientras que la compilación hace casi imposible ver dónde está el error.

4) Un **error sintáctico** ocurre cuando el código fuente no sigue las reglas de la gramática del lenguaje de programación. Esto impide que el compilador o el intérprete entienda la estructura del código y, por lo tanto, **el programa no puede compilarse ni ejecutarse**.
   Ej: (Python)

```
if (x == 10)   # Falta dos puntos al final
   print("x es igual a 10")
```

Un **error semántico** ocurre cuando la estructura del código es correcta desde el punto de vista sintáctico, pero la lógica o el significado de las instrucciones **no tienen sentido** dentro del programa.
Ej: (Python)

```
edad = "veinte"  # Se almacena una cadena en lugar de un número
print(edad + 5)  # Se intenta sumar una cadena y un número
```

5) **Pascal:**

```
Program P
var 5: integer;
var a:char;
Begin
        for i:=5 to 10 do begin
                write(a);
                a=a+1;
        end;
End.
```

- Sintáctico:
  - No se puede declarar una variable con un integer como nombre.
  - a= a+1 está mal, debería ser a:= a+1.
  - Falta el ; en Program P
- Semántico:
  - La variable i no está declarada. Se detecta en compilación
  - No se puede sumar 1 a un char. Se detecta en compilación.
  - “a” no está inicializado, por lo que no se puede imprimir. Se detecta en compilación.

Java:

```
public String tabla(int numero, arrayList<Boolean> listado) {
        String result = null;
        for(i = 1; i < 11; i--) {
                result += numero + "x" + i + "=" + (i*numero) + "\n";
                listado.get(listado.size()-1)=(BOOLEAN) numero>i;
        }
        return true;
}
```

- Lógico:
  - “i” comienza en 1 y resta, por lo que siempre será menor que 11, quedándose en un loop infinito.
- Sintáctico:
  - No es arrayList, sino ArrayList.
  - BOOLEAN es incorrecto, debería ser Boolean.
  - listado.get(listado.size()-1) devuelve un boolean, por lo que no le puede asignar nada.
- Semántico:
  - “i” no está declarado, debe ser int i=1. Se detecta en compilación
  - No puede retornar “true” si debe devolver un String. Se detecta en compilación

- En la primera iteración quiere concatenar un valor a null. Se detecta en compilación.

C:

```
# include <stdio.h>
int suma; /* Esta es una variable global */
int main()
{       int indice;
        encabezado;
        for (indice = 1 ; indice <= 7 ; indice ++)
        cuadrado (indice);
        final(); Llama a la función final */
        return 0;
}
cuadrado (numero)
int numero;
{       int numero_cuadrado;
        numero_cuadrado == numero * numero;
        suma += numero_cuadrado;
        printf("El cuadrado de %d es %d\n",
        numero, numero_cuadrado);
}
```

- Sintácticos:
  - en “encabezado” faltan los paréntesis.
  - Falta abrir el comentario que dice “Llama a la función final */”
  - en “cuadrado(numero)” falta el tipo de función, el ; y las llaves (se usan más abajo de manera incorrecta)
- Semánticos:
  - “encabezado()” no es una función declarada. Se detecta en compilación.
  - “cuadrado(indice)” no es una función declarada. Se detecta en compilación.
  - “final” es una palabra reservada. Se detecta en compilación.
  - “numero” no está inicializado. Se detecta en compilación.
  - “numero_cuadrado” no está inicializado. Se detecta en compilación.
  - “suma” no está declarada.

Python:

```
#!/usr/bin/python
print "\n DEFINICION DE NUMEROS PRIMOS"
r = 1
while r = True:
        N = input("\nDame el numero a analizar: ")
        i = 3
        fact = 0
        if (N mod 2 == 0) and (N != 2):
                print "\nEl numero %d NO es primo\n" % N
        else:
```

```
            while i <= (N^0.5):
                if (N % i) == 0:
                    mensaje="\nEl numero ingresado NO es primo\n" % N
                    msg = mensaje[4:6]
                    print msg
                    fact = 1
                i+=2
                if fact == 0:
                    print "\nEl numero %d SI es primo\n" % N
    r = input("Consultar otro número? SI (1) o NO (0)--->> ")
```

- Sintáctico:
  - Faltan los paréntesis en el primer print.
  - la comparación en el primer while está implementada como asignación.
  - mod no existe, va %.
  - faltan los paréntesis en “print "\nEl numero %d NO es primo\n" % N”.
  - “while i <= (N^0.5):” las potencias se definen con el operador **.
  - “print msg” faltan los paréntesis.
- Semántico:
  - “N” es usado como un número pero es un String. Se detecta en ejecución.
  - “mensaje="\nEl numero ingresado NO es primo\n" % N“ falta el d%. Se detecta en ejecución.

Ruby:

```
def ej1
Puts 'Hola, ¿Cuál es tu nombre?'
nom = gets.chomp
puts 'Mi nombre es ', + nom
puts 'Mi sobrenombre es 'Juan''
puts 'Tengo 10 años'
meses = edad*12
dias = 'meses' *30
hs= 'dias * 24'
puts 'Eso es: meses + ' meses o ' + dias + ' días o ' + hs + ' horas'
puts 'vos cuántos años tenés'
edad2 = gets.chomp
edad = edad + edad2.to_i
puts 'entre ambos tenemos ' + edad + ' años'
puts '¿Sabes que hay ' + name.length.to_s + ' caracteres en tu nombre, ' + name +
'?' end
```

- Semánticos:
  - “edad” no está declarado. Se detecta en ejecución.
  - no existe “name” es “nom”. Se detecta en ejecución.
- Sintácticos:
  - “Puts” va con p minúscula.
  - “puts 'Mi nombre es ', + nom” hay una coma de más.

- puts 'Mi sobrenombre es 'Juan" o hay una comilla de más o falta un + “.
- puts 'Eso es: meses + ' meses o ' + dias + ' días o ' + hs + ' horas' falta una comilla después de meses.

.Net (C#)

```
using System;
class Program
{
   static void OrdenarArreglo(ref char[] arreglo, int cont)
   {
      bool ordenado;
      char aux;
      do
      {
         ordenado = true;
         for (int i = 0; i < cont - 1; i++)
         {
            if (arreglo[i] > arreglo[i + 1]) // Comparación basada en valores ASCII
            {
               aux = arreglo[i];
               arreglo[i] = arreglo[i + 1];
               arreglo[i + 1] = aux;
               ordenado = false;
            }
         }
      } while (!ordenado);
   }
}
```

6) **self:** En Ruby, self hace referencia al objeto receptor actual, es decir, al objeto sobre el cual se está ejecutando el código en un determinado momento. Puede cambiar de contexto dependiendo de dónde se encuentre siendo utilizado.

   nil: En Ruby, nil es un objeto que representa la ausencia de un valor. Se utiliza para indicar que una variable no tiene ningún valor asignado. Es importante destacar que en Ruby, nil es un objeto de la clase NilClass. Se comporta como un valor falso en contextos booleanos.

7) **null:**
   - null es un valor primitivo en JavaScript que representa la ausencia intencional de cualquier valor o un valor nulo.
   - Se utiliza para indicar explícitamente que una variable no contiene ningún valor o que no apunta a ningún objeto o referencia válida.
   - null es de tipo object, pero es técnicamente un valor primitivo.

   undefined:

- Undefined también es un valor primitivo en JavaScript que se asigna automáticamente a una variable cuando se declara pero no se le asigna ningún valor.
- Representa la ausencia de valor debido a que una variable no ha sido inicializada o a que una propiedad de un objeto no está definida.
- Undefined es una forma de "valor por defecto" en JavaScript.

Diferencias Clave:

Característica	`undefined`	`null`
Asignación automática	Sí (por JavaScript)	No (debe asignarse explícitamente)
Significado	Variable no inicializada o propiedad inexistente	Ausencia intencional de valor
Tipo (`typeof`)	`"undefined"`	`"object"` (error histórico)
Uso común	Valores no asignados, funciones sin `return` , propiedades faltantes	Indicar que un valor está vacío intencionalmente

8) La sentencia **break** se usa en varios lenguajes de programación para salir prematuramente de una estructura de control como bucles (for, while, do-while) o switch. Sin embargo, su comportamiento puede variar ligeramente en cada lenguaje.
C:
   - Puede utilizarse dentro de bucles anidados para salir del bucle más cercano al break.
   - No tiene un efecto fuera del bucle o el switch en el que se encuentra.
   - Si se usa dentro de un switch, interrumpe la ejecución del switch, saliendo del bloque switch.
   - No se puede usar fuera de un bucle o switch.

   PHP:
   - Similar a C, puede salir de bucles anidados, pero se puede indicar con un valor numérico la cantidad de anidaciones a salir.

   Javascript:
   - Se comporta de manera similar a C y PHP, saliendo del bucle más cercano o del switch.
   - Puede salir de bucles anidados, pero solo afecta al bucle más cercano.
   - Puede saltar a una etiqueta por ejemplo.

   Ruby:
   - Puede salir de bucles anidados, afectando solo al bucle más cercano. Además permite que salga si tiene una condición. Ej: break if ().

9) La **ligadura (Binding)** es el proceso mediante el cual se asocia un identificador (como una variable, función o tipo de dato) con una entidad en el programa (como un valor, una dirección de memoria o un tipo de dato).
Importancia de la Ligadura en la Semántica:
   - Define **cómo y cuándo** los identificadores se asocian a sus valores o tipos.
   - Afecta la **comprensibilidad** y **predecibilidad** del código.
   - Influye en la **eficiencia** y **seguridad** de ejecución.

Ligadura estática: En la ligadura estática, la asociación entre el identificador y su valor se determina en tiempo antes de la ejecución y permanece constante durante la ejecución del programa.

- Ejemplo: En un lenguaje como C, cuando declaramos una variable global int x = 5;, la ligadura estática vincula el nombre x con el valor 5 en tiempo de compilación.

Ligadura dinámica: En la ligadura dinámica, la asociación entre el identificador y su valor se determina en tiempo de ejecución y puede cambiar durante la ejecución del programa.

- Ejemplo: En lenguajes con asignación dinámica de memoria como Python, cuando asignamos un valor a una variable x = 5, la ligadura dinámica vincula el nombre x con el valor 5 en tiempo de ejecución. Más tarde, podemos cambiar el valor de x a otro valor.