Práctica 3 CPLP

1) La semántica se refiere al significado de las expresiones y símbolos en un lenguaje. En el contexto de la programación, la semántica se refiere al significado de los programas escritos en un lenguaje de programación, es decir, cómo se interpretan las instrucciones y cómo se relacionan entre sí para producir el comportamiento deseado del programa. En otras palabras, la semántica se centra en qué hace un programa, en contraposición a cómo se hace (sintaxis).

La semántica puede ser estática o dinámica. La semántica estática se refiere a las propiedades de un programa que se pueden determinar en tiempo de compilación, como si una variable está declarada o si una función se llama correctamente. La semántica dinámica se refiere a cómo se comporta un programa en tiempo de ejecución, incluyendo cómo se manejan los errores y cómo se gestionan los recursos.

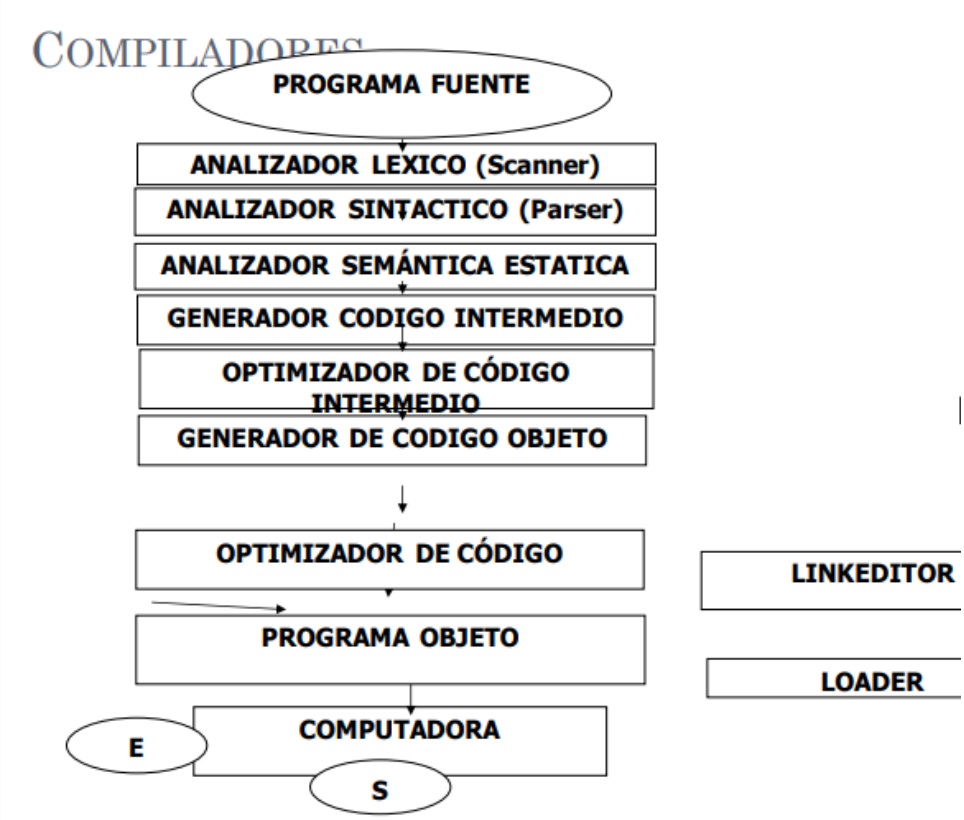
2) a. Compilar un programa significa traducir el código fuente de un programa escrito en un lenguaje de programación de alto nivel a un lenguaje de bajo nivel (código objeto o código binario) que la máquina pueda entender y ejecutar. El proceso de compilación se realiza mediante un programa llamado compilador, que realiza una serie de procesos como análisis léxico, análisis sintáctico y análisis semántico para asegurar que el código fuente sea correcto y válido. El resultado final es un archivo ejecutable que puede ser ejecutado en el sistema operativo correspondiente.

b.La compilación de un programa consta de varios pasos, que pueden variar dependiendo del lenguaje de programación y del compilador utilizado, pero en general, los pasos más comunes son los siguientes:

1. Preprocesamiento: En este paso, el preprocesador toma el código fuente y realiza una serie de transformaciones en él, como incluir las bibliotecas necesarias y expandir las macros definidas.
2. Análisis léxico: El código fuente se divide en una secuencia de tokens, que son unidades sintácticas con un significado semántico específico. Por ejemplo, los tokens pueden ser palabras clave, identificadores, operadores y constantes.
3. Análisis sintáctico: En este paso, el analizador sintáctico verifica que el código fuente cumpla con la gramática del lenguaje de programación. El analizador sintáctico construye un árbol sintáctico que representa la estructura del programa.
4. Análisis semántico: El analizador semántico verifica que el programa sea semánticamente correcto. Por ejemplo, verifica que las variables estén declaradas antes de su uso y que los tipos de datos coincidan en las operaciones.
5. Generación de código intermedio: En este paso, el compilador genera un código intermedio que es más fácil de optimizar y que puede ser utilizado como entrada para diferentes etapas del proceso de compilación.

6. Optimización: En este paso, se realizan una serie de optimizaciones en el código intermedio con el objetivo de mejorar su rendimiento y reducir su tamaño.
7. Generación de código objeto: El código intermedio se traduce en código objeto, que es específico para la arquitectura de la máquina y puede ser ejecutado por ella.
8. Vinculación: Si el programa hace uso de bibliotecas externas, el compilador debe vincular el código objeto generado con las bibliotecas correspondientes para generar el archivo ejecutable final.

En general, estos pasos se llevan a cabo en el proceso de compilación de un programa, aunque la complejidad de cada uno de ellos puede variar dependiendo del lenguaje de programación y del compilador utilizado.



c.La semántica interviene principalmente en la etapa de análisis semántico, que es la tercera etapa de la compilación, después de la etapa de análisis léxico y sintáctico. En esta etapa, el compilador verifica que el programa cumpla con las reglas semánticas del lenguaje, es decir, que el programa tenga sentido en términos de su significado.

La importancia de la semántica en la compilación es fundamental, ya que garantiza que el programa sea coherente y tenga sentido en términos de su significado. Si el compilador encuentra un error semántico, generalmente se genera un mensaje de error que indica que el programa no se puede compilar o que su comportamiento no será el deseado. En general, los errores semánticos son más difíciles de detectar que los errores léxicos o sintácticos, y su corrección puede requerir un análisis más profundo del código.

3) No, no es lo mismo compilar un programa que interpretarlo.

Compilar un programa implica traducir el código fuente escrito en un lenguaje de alto nivel a lenguaje de máquina, lo que da como resultado un archivo ejecutable que se puede ejecutar de forma independiente en el sistema operativo. Este proceso se realiza antes de la ejecución del programa y es necesario hacerlo solo una vez para cada versión del programa.

Por otro lado, interpretar un programa implica ejecutar el código fuente de forma directa en un intérprete, sin necesidad de traducirlo a lenguaje de máquina antes. El intérprete lee y ejecuta el código fuente línea por línea, y lo hace cada vez que se ejecuta el programa. Por lo tanto, el proceso de interpretación es más lento que el proceso de compilación.

Las ventajas de compilar un programa son:

- El programa compilado se puede ejecutar de forma más rápida y eficiente, ya que se ha traducido a lenguaje de máquina de antemano.
- Una vez compilado, el programa se puede ejecutar en cualquier máquina que tenga el mismo sistema operativo y arquitectura de CPU.

Las desventajas de compilar un programa son:

- Es necesario un proceso de compilación adicional antes de que se pueda ejecutar el programa.
- Si se desea ejecutar el programa en diferentes sistemas operativos o arquitecturas de CPU, se deben compilar versiones separadas del programa para cada uno.

Las ventajas de interpretar un programa son:

- El proceso de interpretación permite la ejecución del código fuente en diferentes plataformas sin necesidad de compilarlo para cada una.
- Permite la ejecución del código fuente de forma interactiva, lo que puede ser útil para la depuración y la experimentación.

Las desventajas de interpretar un programa son:

- El proceso de interpretación es más lento que la ejecución de un programa compilado, ya que el intérprete debe leer y ejecutar el código fuente línea por línea.
- No se puede generar un archivo ejecutable para la distribución del programa, lo que hace que el proceso de distribución sea más complicado.

4) Un error sintáctico es un error que ocurre cuando se viola la estructura gramatical del lenguaje de programación. Por lo general, estos errores son detectados por el compilador y se presentan como mensajes de error que indican que la sintaxis del programa es incorrecta. Ejemplos de errores sintácticos incluyen la falta de una llave de cierre, el uso de una variable no declarada o la omisión de un punto y coma al final de una instrucción.

Por otro lado, un error semántico ocurre cuando la lógica del programa es incorrecta o no produce los resultados esperados. Estos errores no son detectados por el compilador y pueden ser más difíciles de encontrar. Ejemplos de errores semánticos incluyen el uso

incorrecto de una función, la asignación incorrecta de una variable o la falta de una condición de salida en un bucle.

En resumen, un error sintáctico es un error de sintaxis o estructura en el código, mientras que un error semántico es un error en la lógica o significado del código. Los errores sintácticos son detectados por el compilador, mientras que los errores semánticos pueden ser más difíciles de detectar y solucionar.

5)

a) 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 Falta el ; , se detecta en compilación

No se puede asignar a un número como nombre a una variable integer

No esta declarada la variable i.

No esta inicializada la variable a

No se le puede asignar un valor a una variable char.
Deberia ser a:= a+1; .Falta el ; . Se detecta en compilación.

Ayuda: Sintáctico 2, Semántico 3

b) 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: La i en el for siempre sera menor a 11. Por lo que se trata de un loop.

Sintáctico:
_arrayList deberia ser ArrayList
_BOOLEAN deberia ser Boolean
_listado.get(listado.size()-1) deberia ser una variable

Semántico:
_la variable i no esta inicializada. Compilación
_numero > i. No se puede transformar un integer en un boolean. Se detecta en compilacion.
_return true: ya se habia especificado que se iba a devolver un String. Compilacion.

Ayuda:
Sintácticos 4, Semánticos 3, Lógico 1

c) 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);
}
```

Sintáctico:
_ encabezado faltan () si es una función declarada.
_ falta el /* en el comentario de la incocación de final()
seria /*Llama a la función final */
_ las llaves de cuadrado(numero) se encuentran mal colocadas.

Semántico:
_encabezado no esta declarada. Se detecta en compilación
_ cuadrado(indice) no esta declarada puesto que tiene mal las llaves. En compilación.
_ la función final() no esta declarada. Compilación
_numero_cuadrado y numero no estan inicializados
_suma no esta inicializada. Compilación.
_ cuadrado(numero) no esta en ningun bloque porque se podria tratar de una invocación pero no es el caso.

Ayuda: *Sintácticos 2, Semánticos 6*

d)Python

```
#!/usr/bin/python
print "\nDEFINICION 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 los print.
_la condición del primer while deberia ser ==
_el mod en py es %
_en while i<= (N^^0.5) la potencia es con **

Semántico:
_while r = true se esta comparando un entero con un boolean. Compilación.
_N = input(...) se deberia parsear lo ingresado a int con (int) porque posteriormente N se usa como entero. Se detecta en compilación
_ falta el %d en , "\nEl numero ingresado NO es primo\n" % N. Se detecta en ejecución.

Ayuda: *Sintácticos 2, Semánticos 3*

e) 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
```

Sintáctico:
_la mayoria de los puts deberian ser Puts con mayúscula.
_puts 'Mi sobrenombre es 'Juan" falta cerrar el ".

Semántico:
_ edad no esta definida . Compilación.
_ "meses" * 30 no se puede combinar un string con un int.
_ puts 'Eso es: meses + ' meses o ' + dias + ' días o ' + hs + ' horas'
No existen las variables meses y dias. Compilación.
_ la variable name no esta definida. Compilación.

Ayuda: *Semánticos +4*

---

6) En Ruby, **`self`** y **`nil`** son variables predefinidas con semánticas especiales.

`self` es una variable especial que representa el objeto actual en el contexto actual de ejecución. Es decir, **`self`** se refiere al objeto que está recibiendo el mensaje en el momento de la ejecución.

Por ejemplo, si se tiene una instancia de la clase **`Person`**, y se llama a un método **`greet`** en esa instancia, dentro del método **`greet`**, **`self`** se referirá a la instancia actual de **`Person`**. Si se llama a un método de clase en la clase **`Person`**, entonces **`self`** se referirá a la propia clase **`Person`**.

Por otro lado, **`nil`** es un objeto especial que representa la ausencia de valor en Ruby. En términos más precisos, **`nil`** es el único objeto en Ruby que representa la ausencia de valor. En Ruby, **`nil`** se utiliza para indicar que una variable o expresión no tiene valor. Por ejemplo, si se intenta obtener un valor de un arreglo más allá de su índice, se devolverá **`nil`**.

Además, en Ruby, **`nil`** se considera un valor falso, lo que significa que cuando se evalúa una expresión en un contexto booleano y la expresión tiene el valor **`nil`**, se considera que la expresión es falsa. Esto se usa comúnmente en las expresiones condicionales, como en **`if`** o **`while`**.

En resumen, **`self`** es una variable que representa el objeto actual en el contexto actual de ejecución, mientras que **`nil`** es un objeto especial que representa la ausencia de valor. Ambas variables predefinidas tienen semánticas específicas y se utilizan de manera común en el lenguaje Ruby.

7) En JavaScript, **`null`** y **`undefined`** son dos valores especiales que se utilizan para representar la ausencia de valor. Aunque ambos valores se utilizan para representar la falta de valor, tienen semánticas ligeramente diferentes.

- **`null`**: Representa la ausencia de valor intencionalmente asignada por el programador. En otras palabras, si una variable tiene un valor de **`null`**, significa que el programador ha decidido que la variable no tenga un valor. Por ejemplo, si se está trabajando con una base de datos y un campo no tiene un valor específico, se puede establecer como **`null`**.
- **`undefined`**: Representa la ausencia de valor asignada automáticamente por el lenguaje de programación. En otras palabras, si una variable tiene un valor de **`undefined`**, significa que el lenguaje de programación no ha asignado un valor a la variable. Esto puede ocurrir si una variable se ha declarado pero no se ha inicializado, o si se ha intentado acceder a una propiedad que no existe en un objeto.

En resumen, **`null`** se utiliza para representar la ausencia intencional de valor, mientras que **`undefined`** se utiliza para representar la ausencia de valor asignada automáticamente por el lenguaje de programación. Es importante tener en cuenta que ambos valores se evalúan a falso en un contexto booleano, lo que significa que si se utilizan en una expresión condicional como **`if`** o **`while`**, se considerarán como falsos.

8) La sentencia **`break`** se utiliza en varios lenguajes de programación para salir de bucles o estructuras de control. A continuación, se describe su semántica en C, PHP, JavaScript y Ruby:

- En C, la sentencia **`break`** se utiliza para salir de bucles **`for`**, **`while`** y **`do-while`**. Cuando se ejecuta la sentencia **`break`**, el flujo del programa sale del bucle y continúa con la siguiente instrucción después del bucle. Es importante tener en cuenta que **`break`** solo puede usarse dentro de un bucle.
- En PHP, la sentencia **`break`** se utiliza para salir de bucles **`for`**, **`foreach`**, **`while`**, y **`do-while`**. Al igual que en C, cuando se ejecuta la sentencia **`break`**, el flujo del programa sale del bucle y continúa con la siguiente instrucción después del bucle. También es importante destacar que **`break`** solo puede usarse dentro de un bucle.
- En JavaScript, la sentencia **`break`** se utiliza para salir de bucles **`for`**, **`while`**, **`do-while`**, y también para salir de la estructura de control **`switch`**. Al igual que en los otros lenguajes, cuando se ejecuta la sentencia **`break`**, el flujo del programa sale del bucle o estructura de control y continúa con la siguiente instrucción después del bucle o estructura de control.
- En Ruby, la sentencia **`break`** se utiliza para salir de bucles **`while`**, **`until`**, **`for`**, y también para salir de la estructura de control **`case`**. Al igual que en los otros lenguajes, cuando se ejecuta la sentencia **`break`**, el flujo del programa sale del bucle o estructura de control y continúa con la siguiente instrucción después del bucle o estructura de control. Es importante tener en cuenta que **`break`** también puede recibir un argumento, que se utiliza para especificar el valor de retorno del bucle.

En resumen, la sentencia **break** se utiliza en varios lenguajes de programación para salir de bucles o estructuras de control. En cada lenguaje, **break** tiene características similares, pero también algunas diferencias en cuanto a los tipos de bucles o estructuras de control en los que se puede utilizar.

9) La ligadura, también conocida como binding, es el proceso de asociar un nombre de variable o función con una entidad específica en el programa, como una dirección de memoria, un valor o una función. La ligadura es un concepto importante en la semántica de un programa, ya que ayuda a determinar el ámbito de una variable o función, así como su tiempo de vida.

Existen dos tipos de ligadura: la ligadura estática y la ligadura dinámica.

La ligadura estática se produce durante la compilación del programa, y se utiliza principalmente en lenguajes de programación de tipado estático, como C o Java. En la ligadura estática, la asociación entre un nombre de variable o función y su entidad correspondiente se establece antes de que el programa se ejecute. Esto significa que el ámbito y el tiempo de vida de la variable o función se pueden determinar antes de que se ejecute el programa.

Un ejemplo sencillo de ligadura estática sería en un programa en C, donde se declara una variable global llamada "contador". La ligadura estática asegura que el nombre "contador" siempre se refiera a la misma dirección de memoria, independientemente de dónde se use en el programa.

Por otro lado, la ligadura dinámica se produce en tiempo de ejecución, y se utiliza principalmente en lenguajes de programación de tipado dinámico, como Python o JavaScript. En la ligadura dinámica, la asociación entre un nombre de variable o función y su entidad correspondiente se establece en tiempo de ejecución, lo que significa que el ámbito y el tiempo de vida de la variable o función se pueden determinar mientras se ejecuta el programa.

Un ejemplo sencillo de ligadura dinámica podría ser en un programa en Python, donde se define una función llamada "suma" que toma dos argumentos. Cuando se llama a la función "suma" con dos números enteros, la ligadura dinámica garantiza que los nombres de los argumentos se asocien con los valores correspondientes durante la ejecución del programa.

En resumen, la ligadura es un concepto fundamental en la semántica de un programa, ya que ayuda a determinar el ámbito y el tiempo de vida de las variables y funciones. La diferencia clave entre la ligadura estática y la ligadura dinámica es el momento en que se establece la asociación entre un nombre y su entidad correspondiente. La ligadura estática se produce durante la compilación del programa, mientras que la ligadura dinámica se produce en tiempo de ejecución.