Conceptos y Paradigmas de Lenguajes de Programación 2025

Práctica Nro. 3
Semántica

Objetivo: Interpretar el concepto de semántica de los lenguajes de programación.

Ejercicio 1: ¿Qué define la semántica?

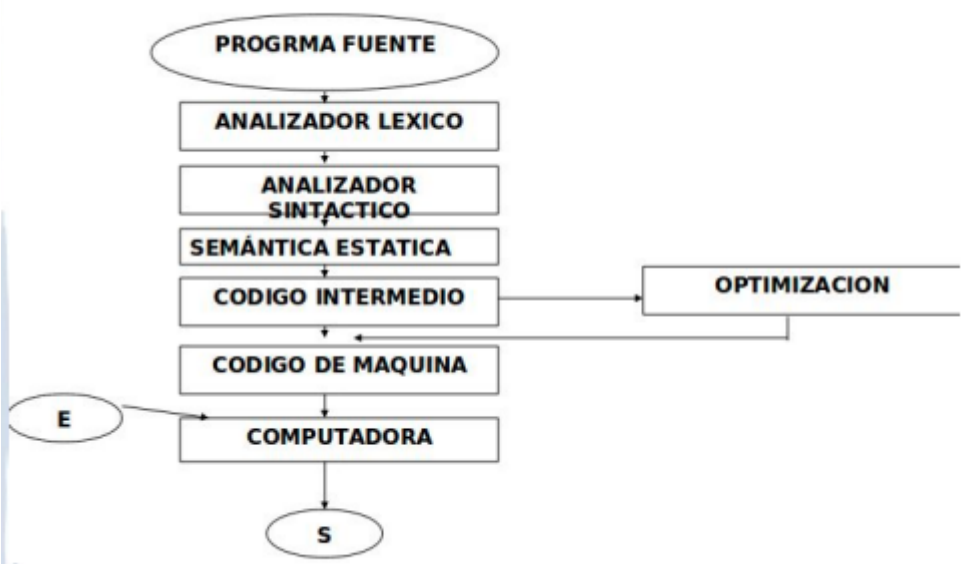
La semántica describe el significado de los símbolos, palabras y frases de un lenguaje ya sea lenguaje natural o lenguaje informático que es sintácticamente válido.

Ejercicio 2:

a. ¿Qué significa compilar un programa?

El compilador es un programa que traduce nuestro programa previo a la ejecución. traduce el programa en el lenguaje máquina para que luego pueda ser ejecutado. El compilador toma todo el programa escrito en un lenguaje de alto nivel que llamamos lenguaje fuente antes de su ejecución. Luego de la compilación va a generar un lenguaje objeto que es generalmente el ejecutable (escrito en lenguaje de máquina .exe) o un lenguaje de nivel intermedio (o lenguaje ensamblador .obj)

b. Describa brevemente cada uno de los pasos necesarios para compilar un programa.



Etapas de análisis (vinculadas al código fuente):

1. Analizador léxico (Scanner) :

- Hace el análisis a nivel de palabra (LEXEMA) y divide el programa en sus elementos/categorías: identificadores, delimitadores, símbolos especiales, operadores, números, palabras clave, palabras reservadas, comentarios, etc. Además, analiza el tipo de cada uno para ver si son TOKENS válidos y filtra comentarios y separadores (como: espacios en blanco, tabulaciones, etc.)
- Es el que lleva más tiempo
- Genera errores si la entrada no coincide con ninguna categoría léxica.
- Reemplaza cada símbolo por su entrada en la tabla de símbolos
- El resultado de este paso será el descubrimiento de los items léxicos o tokens.

2. Analizador sintáctico (Parser):
    - El análisis se realiza a nivel de sentencia/estructuras.
    - Tiene como objetivo identificar las estructuras de las sentencias, declaraciones, expresiones, etc. presentes en su entrada usando los tokens del analizador léxico, estas estructuras se pueden representar mediante el árbol de análisis sintáctico o árbol de derivación, que explica cómo se puede derivar la cadena de entrada en la gramática que especifica el lenguaje, validando qué pertenece o no a la gramática en pos de ver que lo que entra es correcto.
    - El analizador sintáctico (Programa Parser) se alterna/interactúa con el análisis léxico y análisis semántico. Y usualmente usan técnicas de gramáticas formales

3. Analizador semántico (Programa de Semántica estática):
    - Se debe pasar correctamente el Scanner y el Parser.
    - Es una de las etapas más importantes,
    - Se realiza una comprobación de tipos y se agrega información implícita (variables no declaradas).
    - Se hace comprobaciones de nombres y duplicados.
    - Se realizan comproaciones de duplicados, problemas de tipos, comprobaciones de nombres.
    - Nexo entre etapas inicial y final del compilador
    -

Etapa de síntesis (Vinculadas a características del código objeto del hardware y la arquitectura)

4. Optimización del código:
    - No se hace siempre y no lo hacen todos los compiladores.
    - Existen diversas formas en las que se puede optimizar como por ejemplo elegir entre velocidad de ejecución y tamaño del código ejecutable, eliminación de código muerto o no utilizado.
    - Estos programas pueden ser herramientas independientes o estar incluidas en los compiladores y ser invocadas por medio de opciones de compilación
5. Generación del código final
    - El compilador traduce el código fuente y el AST (o alguna representación interna) a código de máquina o código intermedio que la computadora pueda ejecutar

c. ¿En qué paso interviene la semántica y cual es su importancia dentro de la compilación?

La semántica interviene en el análisis semántico (semántica estática) y es de sumamente importancia ya que es el nexo entre las etapas inicial y final del compilador

Ejercicio 3: Con respecto al punto anterior ¿es lo mismo compilar un programa que interpretarlo? Justifique su respuesta mostrando las diferencias básicas, ventajas y desventajas de cada uno.

No, no es lo mismo:

	Compilación	Interpretación
Cómo se ejecuta	Previo a la ejecución el código fuente se traduce completamente a lenguaje de máquina, generando un código objeto de un lenguaje de un modelo mas bajo	El código fuente se traduce línea por línea o instrucción por instrucción mientras se ejecuta. Para ser ejecutado en otra máquina se necesita tener si o si el interprete instalado
Orden en que se ejecuta	Sigue el orden físico de las sentencias.	Sigue el orden lógico de ejecución (ya que va sentencia por sentencia del código)
Tiempo de ejecución	Suele ser mas rápidos en terminos de ejecución ya que el código se traduci directamente a lenguaje maquina. No repite lazos, se decodifica una sola vez.	Suelen ser mas lentos ya que por cada sentencia se realiza el proceso de decodificación para determinas las operaciones a ejecutar y sus operandos. Si la sentencia está en un proceso iterativo, se realiza la tarea tantas veces como sea requerido.
Eficiencia	Pueden realizar optimizaciones durante la compilación, lo que puede resultar en un mejor rendimiento del programa	Pueden ser mas flexibles ya que el código fuente puede ser modificado y ejecutado directamente sin necesidad de recompilación.
Espacio ocupado	Una sentencia puede ocupar decenas o centenas de sentencias de máquina al pasar a código objeto	Ocupa menos espacio de memoria ya que cada sentencia se deja en la forma original y las instrucciones necesarias para ejecutarlas se

		almacenan en los subprogramas del interprete en memoria
Detección de errores	Le cuesta más determinar los errores ya que cualquier referencia al código fuente se pierden en el código objeto. Se pierde la referencia entre el código fuente y el código objeto. Es casi imposible ubicar el error, pobres en significado para el programador.	Las sentencias del código fuente puede ser relacionadas directamente con la que se está ejecutando. Se puede ubicar el error, es más fácil detectarlos por donde pasa la ejecución y es más fácil corregirlos

Ejercicio 4: Explique claramente la diferencia entre un error sintáctico y uno semántico. Ejemplifique cada caso.

Ejercicio 5: Sean los siguientes ejemplos de programas. Analice y diga qué tipo de error se produce (Semántico o Sintáctico) y en qué momento se detectan dichos errores (Compilación o Ejecución). Aclaración: Los valores de la ayuda pueden ser mayores.

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.
```

Ayuda: Sintáctico 2, Semántico 3
Sintácticos:

- falta ; despues de Program P
- var 5: integer: un numero no puede ser el nombre de una variable
- está mal la asignación a=a +1 debería ser a:= a+1

Semánticos:

- La variable i no está declarada
- La variable a no esta inicializada
- La variable a es de tipo char por lo que no se le puede sumar un número

Semanticos

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;
}
```

Ayuda:
Sintácticos 4, Semánticos 3, Lógico 1
Sintácticos:

- Esta mal escrito el tipo arrayList, debería ser ArrayList

Semántico:

- la variable i no está declarada
- no se puede asignar un valor a otro
- no se puede castear un numero a boolean??
- return true ya que la funcion deberia devolver un String
- se inicializa el acumulador result en null y no se puede concatenar con null
-

Lógico:

-Bucle for es infinito

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);
}
```

Ayuda: Sintácticos 2, Semánticos 6

Sintactico:

-no está declarado el tipo de la variable encabezado

-for no tiene las llaves de inicio ni de cierre // Puede que no este mal porque puede ser un for de una linea

-Falta /* al inicio del comentario

-A la funcion cuadrado le faltan las llaves

-La función cuadrado no especifica el tipo de retorno o void

-falta la llave de apertura en el for

-numero_cuadrado == numero*numero deberia ser asignacion(=) y es una comparación(==)

Semantico:

- El for llama a la función cuadrado que no está declarada en ese contexto

- La función final() no está declarada

-La función cuadrado usa la variable numero que no se encuentra declarada previa a la declaración de la función

-suma +=numero_cuadrado : se asigna una variable que no esta inicializada

Logico:

-el for nunca termina

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)--->> ")
```

Sintactico

- El print lleva paréntesis
- while r = true deberia ser comparacion (==)
- El operador ‘mod’ se escribe con %

Semantico:

- El input N necesita un casteo a int
- `mensaje="\nEl numero ingresado NO es primo\n" % N` : hace falta %d
-
-

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
```

Ayuda: Semánticos +4

Sintáctico:

- el primer Puts esta en mayuscula y da el error: undefined method `Puts'
- `puts 'Mi nombre es ', + nom` : la coma va adentro de las comillas
- `dias = 'meses' *30` : no se puede multiplicar strings
- `'Eso es: meses + ' meses o '` la concatenación quedó adentro de las comillas

Semántico:

- La variable ‘edad’ no está declarada, tira el error: undefined local variable or method `edad' for main:Object (NameError)
- La variable name no está declarada

Ejercicio 5:Dado el siguiente código escrito en pascal. Transcriba la misma funcionalidad de acuerdo al lenguaje que haya cursado en años anteriores. Defina brevemente la sintaxis (sin hacer la gramática) y semántica para la utilización de arreglos y estructuras de control del ejemplo

```
Procedure ordenar_arreglo(var arreglo: arreglo_de_caracteres;cont:integer);
var
        i:integer; ordenado:boolean;
        aux:char;
begin
        repeat
        ordenado:=true;
        for i:=1 to cont-1 do
        if ord(arreglo[i])>ord(arreglo[i+1])
                then begin
                        aux:=arreglo[i];
                        arreglo[i]:=arreglo[i+1];
                        arreglo[i+1]:=aux; ordenado:=false
                        end;
        until ordenado;

end;
```

Observación: Aquí sólo se debe definir la instrucción y qué es lo que hace cada una; detallando alguna particularidad del lenguaje respecto de ella. Por ejemplo el for de java necesita definir una variable entera, una condición y un incremento para dicha variable.

Java

```
   public static void ordenarArreglo(char[] arreglo) {
      int cont = arreglo.length;  // Obtiene el tamaño del arreglo
      boolean ordenado;
      char aux;

      // Bucle para ordenar los elementos (algoritmo burbuja)
      do {
         ordenado = true;  // Asumimos que el arreglo está ordenado
         for (int i = 0; i < cont - 1; i++) {
            // Comparar los valores adyacentes
            if (arreglo[i] > arreglo[i + 1]) {
               // Intercambiar si están en el orden incorrecto
               aux = arreglo[i];
               arreglo[i] = arreglo[i + 1];
               arreglo[i + 1] = aux;
               ordenado = false;  // Indicar que el arreglo no está ordenado
            }
         }
      } while (!ordenado);  // Repetir mientras no esté ordenado
   }

   public static void main(String[] args) {
      char[] arreglo = {'z', 'a', 'd', 'b', 'c'};

      // Llamada al procedimiento para ordenar el arreglo
      ordenarArreglo(arreglo);

      // Mostrar el arreglo ordenado
      for (char c : arreglo) {
         System.out.print(c + " ");
      }
   }
}
```

Ejercicio 6: Explique cuál es la semántica para las variables predefinidas en lenguaje Ruby **self** y **nil**. ¿Qué valor toman; cómo son usadas por el lenguaje?

self en Ruby:

- self es una variable especial en Ruby que hace referencia al objeto actual en el contexto donde se usa. Dependiendo de dónde se utilice, el valor de self puede cambiar. Es decir, puede hacer referencia a un objeto distinto en distintos lugares del código.
- Dentro de un método de instancia: self hace referencia al objeto en el que se está invocando el método.
- Dentro de un método de clase: self hace referencia a la clase en sí misma. Dentro de un bloque o dentro de un contexto sin un objeto explícito: self puede ser el objeto main (el entorno global), si no hay un objeto específico al que hacer referencia.

nil en Ruby:

- nil es un objeto en Ruby que representa la ausencia de valor o un valor nulo. Es similar a null en otros lenguajes de programación.
- Es un objeto especial de la clase NilClass, y tiene su propio significado dentro del lenguaje.
- nil es el valor que se devuelve cuando algo no tiene un valor asignado explícitamente.
- Por defecto, cuando un método no devuelve nada explícitamente, su valor de retorno es nil.

Ejercicio 7: Determine la semántica de null y undefined para valores en javascript.¿Qué diferencia hay entre ellos?

En JavaScript, tanto `null` como `undefined` son valores especiales que representan "ausencia de valor", pero tienen diferencias en su semántica, uso y contexto:

null

- `null` es un valor primitivo en JavaScript que representa la ausencia deliberada de un valor. Es un objeto que se utiliza cuando se quiere que una variable tenga un valor "vacío", es decir, un valor que ha sido explicitamente asignado para indicar la ausencia de un objeto.
- En términos de semántica, `null` se usa para indicar que una variable tiene intencionadamente un valor nulo. Es comúnmente utilizado cuando deseas que una variable no apunte a ningún objeto, pero aún se haya inicializado.

undefined

- undefined es un valor primitivo en JavaScript que indica que una variable ha sido declarada pero no se le ha asignado ningún valor aún. También es el valor de retorno de una función que no tiene una declaración explícita de return. En otras palabras, undefined es el valor por defecto de las variables no inicializadas.

	Null	**Undefined**
Característica	Valor asignado explícitamente para indicar la ausencia de un valor.	Valor asignado implícitamente a una variable no inicializada o función sin valor de retorno.
Tipo de dato	object (aunque esto es históricamente un error en JavaScript)	undefined
Asignación	Se asigna explícitamente por el programador.	Se asigna automáticamente por JavaScript si una variable es declarada pero no inicializada.
Uso común	Usado para indicar la ausencia de un objeto o valor.	Usado cuando una variable o propiedad no ha sido asignada o una función no tiene un valor de retorno.
Comparación	null === null es true, null === undefined es false	undefined === undefined es true, undefined === null es false

Ejercicio 8: Determine la semántica de la sentencia break en C, PHP, javascript y Ruby. Cite las características más importantes de esta sentencia para cada lenguaje

La sentencia break se utiliza para interrumpir el flujo de un ciclo o una estructura de control de bucles (como for, while, do-while), y en algunos lenguajes también se puede usar en switch/case

1. C:

Semántica:

En C, la sentencia break se usa para salir de un bucle o de un bloque switch/case antes de que se complete su ejecución.

Características importantes:

- Bucle: En un for, while, o do-while, break termina el ciclo de manera inmediata y salta a la siguiente instrucción después del bucle.

- Switch/case: En una estructura switch, break se utiliza para salir del switch y evitar que el control pase a los casos siguientes.

2. PHP:

Semántica:

En PHP, break funciona de manera similar a C. Sirve para salir de un ciclo o de un bloque switch/case. Además, permite especificar un nivel de anidamiento en el que se desea realizar la interrupción (esto es útil cuando tenemos bucles anidados).

Características importantes:

- Bucle: Interrumpe el ciclo for, while, do-while.
- Switch/case: Termina el bloque switch/case.
- Anidamiento: Puede aceptar un parámetro que indique cuántos niveles de bucles anidados deben ser interrumpidos.

3. JavaScript:

Semántica:

En JavaScript, break tiene un funcionamiento muy similar al de C. Se usa para salir de un bucle o de una estructura switch/case.

Características importantes:

- Bucle: Se utiliza dentro de ciclos for, while, y do-while para salir del ciclo antes de que termine su ejecución.
- Switch/case: En un bloque switch, break se usa para evitar que el control pase a los siguientes casos.

4. Ruby:

Semántica:

En Ruby, break también interrumpe la ejecución de un bucle, pero también se puede usar dentro de bloques. Esto significa que no solo interrumpe ciclos tradicionales como for, while, y each, sino que también puede salir de cualquier tipo de bloque (como los bloques de métodos que usan yield).

Características importantes:

- Bucle: Se utiliza dentro de ciclos for, while, y until para salir del ciclo.
- Bloques: Ruby permite salir de bloques con break, lo que es útil cuando se usan métodos que iteran sobre colecciones.
- Valor de retorno: Ruby permite devolver un valor con break al salir del bucle (esto es distinto a otros lenguajes como C o JavaScript).

Lenguaje	Función básica de `break`	Características adicionales
C	Salir de ciclos (for, while, do-while) y bloques switch/case.	No permite especificar niveles de anidamiento.
PHP	Salir de ciclos y bloques switch/case.	Puede aceptar un parámetro (`break n`) para salir de varios niveles de bucles.
JavaScript	Salir de ciclos (for, while, do-while) y bloques switch/case.	No permite especificar niveles de anidamiento.
Ruby	Salir de ciclos y bloques	Permite devolver un valor con `break`, útil para iteraciones

Ejercicio 9:Defina el concepto de ligadura y su importancia respecto de la semántica de un programa. ¿Qué diferencias hay entre ligadura estática y dinámica? Cite ejemplos (proponer casos sencillos)

Concepto de ligadura en programación

La ligadura (también conocida como binding) se refiere al proceso de asociar una variable o un identificador con un valor o una ubicación en la memoria en el contexto de un programa. La ligadura determina cuándo y cómo se asigna el valor a una variable, y puede ocurrir en diferentes momentos durante la ejecución de un programa.

Dependiendo del momento en que se realice la ligadura, la ligadura puede ser estática o dinámica.

Importancia de la ligadura en la semántica de un programa

La ligadura es fundamental porque influye directamente en la resolución de los valores de las variables en tiempo de ejecución. Una ligadura incorrecta o mal gestionada puede causar errores en el programa, como referencias incorrectas a variables o valores no esperados.

Conceptos y Paradigmas de Lenguajes de Programación 2025

Ligadura Estática (o ligada en tiempo de compilación)

La ligadura estática se refiere a la resolución de las variables y sus valores en el momento en que el programa es compilado. Es decir, el sistema sabe desde el principio qué valores estarán asociados con las variables a lo largo de la ejecución del programa.

Características:

- Se determina en tiempo de compilación.
- No cambia durante la ejecución del programa.
- Se asocia a la resolución de las variables antes de que se ejecute el programa, lo que significa que el tipo y la memoria de las variables son fijos.

Ventajas:

- Más eficiente: Se puede hacer una optimización del código, ya que el sistema sabe todo en el momento de la compilación.
- Seguridad de tipo: El tipo de las variables es conocido, lo que reduce la posibilidad de errores en la ejecución.

Desventajas:

- Falta de flexibilidad: No se pueden hacer modificaciones a las variables de manera dinámica según el contexto de ejecución.

Ligadura Dinámica (o ligada en tiempo de ejecución)

La ligadura dinámica se refiere a la resolución de las variables y sus valores en el momento en que el programa está siendo ejecutado. Esto significa que el valor de una variable puede cambiar dependiendo del contexto de ejecución, y no está fijado de antemano.

Características:

- Se determina en tiempo de ejecución.
- Permite asociaciones flexibles que pueden cambiar durante la ejecución del programa.
- Es común en lenguajes que usan tipado dinámico (por ejemplo, Python, JavaScript).

Ventajas:

- Flexibilidad: Permite que las variables se resuelvan en función del contexto de ejecución.
- Programación más dinámica: Es útil para escribir programas más generales o interactivos donde los valores de las variables pueden cambiar en tiempo de ejecución.

Desventajas**:**

- Menos eficiente: Como las decisiones sobre las variables se toman en tiempo de ejecución, el sistema no puede optimizar el código como lo haría con ligadura estática.
- Mayor riesgo de errores: Al depender de la ejecución, pueden surgir errores inesperados si el valor de las variables cambia de manera inesperada

NOTAS 4/4
Conceptos de ligadura

- Alcance: donde se conoce la variable
- Tiempo de vida: tiempo en que esta alocada en memoria
- Super importante: Diferencia entre esas dos

R Valor: El dato que tiene (el valor codificado que tiene almacenado en la memeoria)

Ligadura:
- Es estática si se establece antes de la ejecución
- Es dinámica cuando se hace durante la ejecución

Tipo: Define el conjunto de valores posibles que puede tomar y las operación que puedo realizar sobre esa variable

Variables estáticas(que su tiempo de vida va seguir existiendo aun cuando el programa no se ejecute más): Según la cátedra: SOLO EN C
Variables dinámicas: se generar dos variables: El puntero y a quien apunta el puntero(esta es la verdadera variable dinámica)
Semi Dinámicas (arreglos semidinámicos): Es un arreglo que puedo declarar e inicializar sin especificar la cantidad de elementos que va a tener. (SOLO ESTÁ EN ADA)
R-valor depende de con qué valor el lenguaje inicializa esa variable. Ejemplo en C las inicializa en 0
Una variable estática en java se considera automática según la catedra.
<1-10>:significa existia antes del programa (<)y despues de la linea 10 va a seguir existiendo, es decir, después de que termine el programa

Enmascarar una variable: Dos variables se llaman igual pero tienen distintos alcances. Ejemplo: tengo una que tiene alcance global y no quiero usarla en un método entonces en ese método declaro una con el mismo nombre entonces cuando busque a la variable en ese método voy a usar la local, no la global.

Conceptos y Paradigmas de Lenguajes de Programación 2025

Extern en C: Extern int v1 → significa que hace referencia a otra variable v1 que está definida en otro lado.

1a) Variable enmascarada
1b) Variable dinámica que le hice dispose (acorte el tiempo de vida)
1c) Una variable automática que sea igual que su alcance