

Practica 3 - Semantica

Ejercicio 1: ¿Qué define la semántica?

La semantica define el **significado de las construcciones del lenguaje**. Mientras que la sintaxis describe como deben **estructurarse correctamente las expresiones**, la semantica especifica que ocurre cuando se ejecutan.

Sintaxis: Define como debe escribirse el codigo correctamente, pero NO indica el significado.

Semantica: Define que hace realmente el codigo cuando se ejecuta

Ejemplo de semantica vs sintaxis...

Sintaxis de una asignacion en java :

```
int x = 5;
```

Semantica: Se reserva un espacio de memoria para la variable `x` y se le asigna un valor `5`

Ejercicio 2:

a. ¿Qué significa compilar un programa?

Compilar un programa significa traducir el codigo fuente escrito en un lenguaje de alto nivel (como C, java, pyhton) a un lenguaje de maquina o codigo intermedio que la computadora pueda ejecutar. Este proceso es realizado por un compilador, que analiza el codigo, detecta errores y lo transforma en un formato ejecutable.

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

Primer paso: Etapa de analisis:

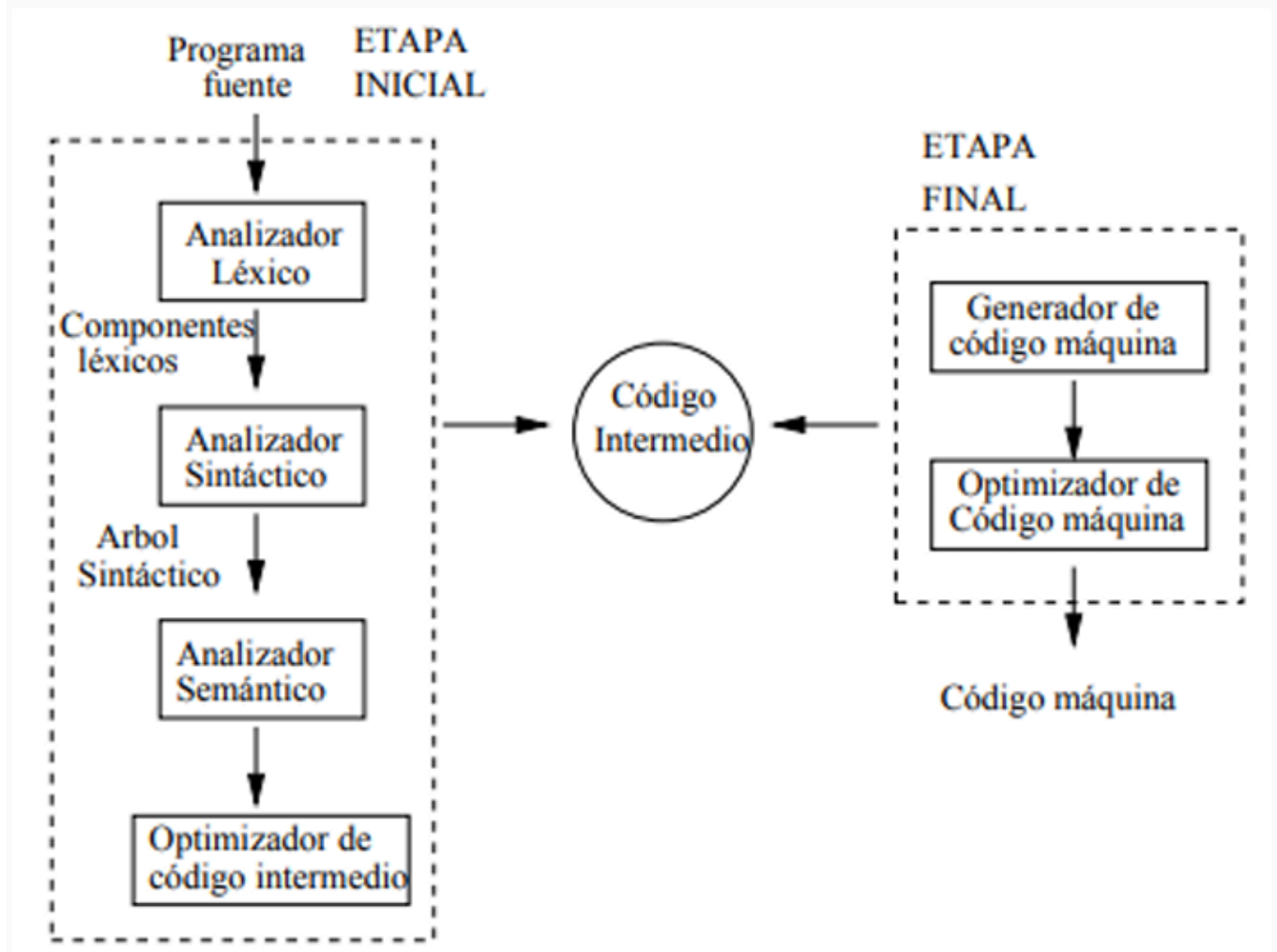
- Analisis lexico:
 - El compilador divide el codigo en tokens (palabras clave, identificadores, operadores, etc)
 - Detecta errores como caracteres invalidos o estructuras incorrectas
- Analisis sintactico:

- Verifica si la estructura del código sigue las reglas gramaticales del lenguaje
- Genera un árbol de sintaxis abstracta.
- Detecta errores como parentesis mal cerrados o estructuras incorrectas.
- Analisis semantico: Debe pasar bien el analisis lexico y el analisis sintactico. **Procesa** las estructuras sintacticas reconocidas por el analizador sintactico.
 - Verifica tipos de datos, alcance de variables y reglas logicas.
 - Ejemplo de error semantico: Intentar sumar un numero con un string sin conversion explicita.
- Generar código intermedio:
 - Implica realizar la transformacion del código fuente en una representacion de código intermedio para una maquina abstracta.

Segundo paso: Etapa de sintesis:

- Construye el programa ejecutable y genera el código necesario.
- Se realiza el proceso de optimizacion (optativo)
-

COMPILADORES – CÓMO FUNCIONAN



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

La semántica interviene en el análisis semántico (tercer paso). Su importancia radica en asegurar que el código no solo sea sintacticamente correcto, sino que también tenga sentido lógico.

```
int x = "Hola"; // Error semantico: No se puede asignar un string a un int.
```

En este ejemplo, la sintaxis es válida, pero la semántica dentro del lenguaje Java por ejemplo, impide esa asignación. Si el tipo de la variable es `int`, solo permite valores de tipo Enteros.

Sin un análisis semántico, el programa podría compilarse con errores lógicos graves, generando comportamientos inesperados o fallas en tiempo de ejecución.

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 es lo mismo compilar un programa que interpretarlo, son dos conceptos totalmente distintos.

Compilar: Traduce todo el código fuente a un lenguaje máquina (binario) antes de ejecutarlo. Los errores se detectan antes de la ejecución, en la fase de compilación.

Interpretar: No traduce el código completo, sino que lo lee y lo ejecuta línea por línea. Traduce la línea a lenguaje máquina y la ejecuta inmediatamente. Repite el proceso hasta completar la ejecución del programa. Los errores se detectan en tiempo de ejecución, lo que puede causar fallos inesperados.



Diferencia clave:


- **Compilado:** El código se traduce una sola vez y luego se ejecuta.
- **Interpretado:** Se traduce y ejecuta al mismo tiempo, línea por línea.
 - 📌 **Ejemplo de lenguajes compilados:** C, C++, Rust.
 - 📌 **Ejemplo de lenguajes interpretados:** Python, JavaScript, Bash.

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



Diferencias entre error sintáctico y error semántico

Tipo de Error	Definición
Error sintáctico	<p>Se produce cuando el código no cumple con las reglas de sintaxis del lenguaje. Es como un error gramatical en un idioma.</p> <p>Ejemplo en java:</p> <pre>int a:= 3; // ❌ ERROR</pre> <p>En java, la operación de asignación, se lleva a cabo mediante el '='</p> <pre>int a = 3; // ✅ **Corrección:**</pre>
Error semántico	<p>Ocurre cuando el código es válido sintácticamente pero su significado es incorrecto, causando fallos en la lógica del programa.</p> <p>Ejemplo en java:</p>

Tipo de Error	Definición
	<pre>int b = "Hola"; // ❌ ERROR</pre> <p>En java, la sintaxis es valida, pero el significado es incorrecto porque <code>int</code> no puede almacenar valores de tipo <code>String</code></p> <pre>String b = "Hola";//✅ **Corrección:**</pre>
 Conclusión	<p>Error sintáctico: Problema con la estructura del código, detectado en la compilación.</p> <p>Error semántico: Problema con la lógica del código, detectado en la ejecución o en análisis de tipos.</p>

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)

A)

```
Pascal
Program P // (1)
var 5: integer; // (2)
var a:char;
Begin
    for i:=5 to 10 do // (3)
        begin
            write(a); // (4)
            a=a+1; // (5)
        end;
End.
```

Ayuda: Sintáctico 2, Semántico 3

Errores:

(1) Error sintactico: Falta ' ; '

(2) Error sintactico,: Una variable no puede comenzar con numero

(3) Error semantico, La variable 'i no se encuentra declarada

(4) Error semantico: La variable 'a' no esta inicializadaa.

(5) Error sintactico: La asignacion se hace con ' := '

(5) Error semantico: No se le puede sumar 1 a una variable de tipo char.

B) Java:

```
public String tabla(int numero, arrayList<Boolean> listado) // (1)
{
    String result = null;
    for(i = 1; i < 11; i--) { // (2)
        result += numero + "x" + i + "=" + (i*numero) + "\n"; // (3)
        listado.get(listado.size()-1)=(BOOLEAN) numero>i; // (4)
    }
    return true; // (5)
}
...
```

Ayuda:

Sintácticos 4, Semánticos 3, Lógico 1

Errores:

(1) Error sintactico: arrayList este tipo de dato no existe, deberia ser ArrayList, con la 'A'

(2) Error semantico: La variable ' i ' no se encuentra declarada

(2) Error logico: La condicion va a ser siempre verdadera, se genera bucle infinito.

(3) Error semantico: Concatenar cadenas con null genera nullPointerException.

(4) Error sintactico: La asignacion es invalida.

(5) Error sintactico: El metodo intenta devolver true (boolean), pero el timpo de dato que retorna el metodo es un String.

c , d y e como no conozco tanto los lenguajes, los voy a dejar para mas adelante...

Ejercicio 5:

```java

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;

```

Tengo dudas con esta resolucion que viene a continuacion porque en java en vez de usar [] podemos usar ArrayList que nos proporciona metodos para ordenar, por ejemplo podemos usar el metodo sort.

Pero no se si quieren que pasemos la sintaxis con la misma semantica a Java, o que hagamos el mismo metodo pero orientado a programacion en java (objetos), por ejemplo usando objetos como ArrayList para poder acceder a sus metodos

Solucion manteniendo la misma semantica:

Lo que esta marcado entre parentesis, son diferencias sintacticas.

```

public void ordenar_arreglo(char[] arregloDeCaracteres, int cont) // (1)
{ //(2)
 boolean ordenado;
 char aux;
 do {
 ordenado = true; // (4)
 for (int i = 0; i < cont - 1; i++) //(3)
 {
 if (arregloDeCaracteres[i] > arregloDeCaracteres[i + 1]) {
 aux = arregloDeCaracteres[i];
 arregloDeCaracteres[i] = arregloDeCaracteres[i + 1];
 arregloDeCaracteres[i + 1] = aux;
 ordenado = false;
 }
 }
 } while (!ordenado);
}

```

Diferencias sintacticas:

(1): En java, se necesita un modificador de acceso para el metodo, ademas del tipo de dato que devuelve.

Como el metodo no devuelve nada, se dice expresa como 'void'

(2): En java no tenemos una zona especifica para declaracion de variables, si no que las podemos declarar en cualquier momento

(3): En java, la estructura for, es necesario expresar la variable y el valor de inicio, la condicion final del bucle, y la forma en se que incrementa

(4): En java, la asignacion se hace mediante '=' a diferencia de pascal que es necesario ':='

Diferencias semanticas :

- Diferencias en indices de arreglos:

En java, todos los arreglos comienzan necesariamente desde la posicion 0.

Mientras que en Pascal, no necesariamente pueden arrancar de la posicion 0, ya que se puede definir las posiciones de inicio y fin.

- Diferencias en estructuras de control:

Pascal usa repeat untill (condicion), que ejecuta primero el codigo y luego evalua la condicion.

Java no tiene un repeat untill, sino que se traduce a un do while con la misma logica.

Solucion adaptada a Java (Se utiliza ArrayList para aprovechar el metodo Collections.sort())

Veo innecesario llamar a un metodo que es ordenar\_arreglo, cuando esto se podria hacer en la misma linea utilizando el Collectoion

```
public void ordenar_arreglo(ArrayList<Character> arregloDeCaracteres)
{
 Collections.sort(arregloDeCaracteres);
}
```

## Ejercicio 6:

En Ruby, las variables `self` y `nil` tienen significados específicos y juegan un papel clave en la semántica del lenguaje.

### ◆ `self` en Ruby

- `self` representa **el objeto actual en contexto**.
- En métodos de instancia, `self` se refiere al objeto que llama al método.
- En métodos de clase, `self` representa la clase misma.
- Se usa para acceder a atributos, métodos y redefinir métodos dentro del objeto actual.

```
class Persona
 attr_accessor :nombre
```



```

def initialize(nombre)
 self.nombre = nombre # Usa self para llamar al setter
end

def mostrar_self
 puts "El self dentro de esta instancia es: #{self}"
end

p = Persona.new("Juan")
p.mostrar_self # Muestra el objeto Persona en el que estamos trabajando

```

## ◆ nil en Ruby

- `nil` representa **la ausencia de un valor**.
- Es el único objeto de la clase `NilClass` en Ruby.
- Equivale a `null` en otros lenguajes como Java o C#.
- En condiciones, `nil` se evalúa como `false`.

```

valor = nil
puts "La variable está vacía" if valor.nil? # Devuelve true

```

## Resumen:

| Variable          | Significado                 | Uso                                                            |
|-------------------|-----------------------------|----------------------------------------------------------------|
| <code>self</code> | Representa el objeto actual | Acceso a atributos/métodos dentro de un objeto                 |
| <code>nil</code>  | Ausencia de valor           | Representa un objeto vacío y se evalúa como <code>false</code> |

### Ejercicio 7:

En JavaScript, tanto `null` como `undefined` representan la ausencia de valor, pero tienen diferencias clave en su uso y significado.

## ◆ undefined

- Se asigna automáticamente a variables que no han sido inicializadas.
- También se obtiene cuando se intenta acceder a propiedades o elementos inexistentes.

- Es el valor por defecto de parámetros en funciones si no se les pasa un argumento.
- Representa **"ausencia de valor por omisión"**.

```
let x;
console.log(x); // undefined (no se ha inicializado)

function saludar(nombre) {
 console.log("Hola " + nombre);
}
saludar(); // Hola undefined (no se pasó argumento)

let obj = {};
console.log(obj.propiedadInexistente); // undefined
```

## ◆ null

- Es un valor explícito asignado a una variable para indicar **"ausencia intencional de valor"**.
- Se usa cuando se quiere dejar claro que una variable no tiene ningún valor válido.
- Es un objeto en JavaScript ( `typeof null` devuelve `"object"` , aunque esto es un error histórico).

```
let y = null;
console.log(y); // null (ausencia intencional de valor)

let usuario = { nombre: "Ana" };
usuario = null; // Se borra la referencia al objeto
```

## ◆ Diferencias entre null y undefined

| Característica | undefined                                                                    | null                                        |
|----------------|------------------------------------------------------------------------------|---------------------------------------------|
| Asignación     | Automática cuando una variable no está definida                              | Se asigna intencionalmente                  |
| Significado    | "No definido"                                                                | "Ausencia de valor"                         |
| Tipo de dato   | undefined                                                                    | object (error histórico en JS)              |
| Uso común      | Variables no inicializadas, propiedades faltantes, parámetros sin argumentos | Representar valores vacíos intencionalmente |

## Conclusión:

- Usa `undefined` cuando algo no está definido naturalmente en JavaScript.
- Usa `null` cuando quieres representar intencionalmente que no hay un valor válido.

## Ejercicio 8:

La sentencia `break` se usa en los lenguajes mencionados en el enunciado para **salir de un bucle o estructura de control antes de que termine naturalmente**. Sin embargo, hay diferencias en su uso y comportamiento en cada lenguaje.

### break en C

#### Características:

- Se usa en bucles (`for`, `while`, `do-while`) para terminar la ejecución antes de cumplir la condición de salida.
- También se usa en estructuras `switch` para salir de un caso y evitar la ejecución en cascada (`fall-through`).
- No permite salir de múltiples niveles de bucle directamente (se puede usar `goto` para eso).

```
#include <stdio.h>
int main() {
 for (int i = 0; i < 10; i++) {
 if (i == 5) {
 break; // Sale del bucle cuando i es 5
 }
 printf("%d\n", i);
 }
 return 0;
}
```

### break en PHP

#### Características:

- Funciona de manera similar a C y JavaScript.
- Se usa en `for`, `while`, `do-while`, `foreach` y `switch`.
- PHP permite `break N`, donde `N` indica cuántos niveles de bucle salir.

```
for ($i = 0; $i < 10; $i++) {
 if ($i == 5) {
 break; // Sale del bucle cuando $i es 5
 }
}
```

```
}
 echo "$i\n";
}
```

## ◆ break en JavaScript

### 📌 Características:

- Se usa en bucles ( `for` , `while` , `do-while` ) y `switch` .
- No tiene `break N` como en PHP, pero se puede usar etiquetas ( `labels` ) para salir de múltiples niveles de bucles.

```
for (let i = 0; i < 10; i++) {
 if (i === 5) {
 break; // Sale del bucle cuando i es 5
 }
 console.log(i);
}
```

## ◆ break en Ruby

### 📌 Características:

- Se usa en bucles ( `while` , `until` , `for` ) y dentro de iteradores ( `each` , `loop` ).
- Cuando `break` se ejecuta, el bucle termina inmediatamente y se devuelve `nil` .

```
for i in 0..9
 if i == 5
 break # Sale del bucle cuando i es 5
 end
 puts i
end
```

## ◆ Comparación de break en los lenguajes

| Lenguaje | Uso en bucles                                                                          | Uso en <code>switch</code> | Soporte para múltiples niveles   |
|----------|----------------------------------------------------------------------------------------|----------------------------|----------------------------------|
| C        | ✅ <code>for</code> , <code>while</code> , <code>do-while</code>                        | ✅ Sí                       | ❌ No (se usa <code>goto</code> ) |
| PHP      | ✅ <code>for</code> , <code>while</code> , <code>do-while</code> , <code>foreach</code> | ✅ Sí                       | ✅ Sí ( <code>break N</code> )    |

| Lenguaje | Uso en bucles                                                                    | Uso en <code>switch</code>           | Soporte para múltiples niveles          |
|----------|----------------------------------------------------------------------------------|--------------------------------------|-----------------------------------------|
| JS       | ✅ <code>for</code> , <code>while</code> , <code>do-while</code>                  | ✅ Sí                                 | ✅ Con etiquetas ( <code>labels</code> ) |
| Ruby     | ✅ <code>for</code> , <code>while</code> , <code>until</code> , <code>each</code> | ❌ No hay <code>switch</code> clásico | ❌ No (sólo sale del bloque actual)      |

### Conclusión:

- `break` se usa en todos los lenguajes para **salir de un bucle o `switch`**.
- PHP y JavaScript permiten salir de múltiples bucles ( `break N` y `labels` respectivamente).
- Ruby permite `break` en iteradores como `each`.
- C no permite `break` en múltiples niveles sin `goto`.

## Ejercicio 9:

### ◆ Definición:

La **ligadura** (o **binding**) es el proceso mediante el cual se asocia un identificador (como una variable, función o método) con una entidad concreta en la memoria o en el código.

### ◆ Importancia:

La ligadura es crucial para definir el **comportamiento semántico** de un programa, ya que determina cuándo y cómo se resuelven los nombres de variables, funciones y objetos. Esto influye en aspectos como el rendimiento, la flexibilidad y la detección de errores en tiempo de compilación o ejecución.

## Ligadura Estática vs. Dinámica

| Tipo de Ligadura                | ¿Cuándo se resuelve?            | Características                    | Ejemplo de uso                                                            |
|---------------------------------|---------------------------------|------------------------------------|---------------------------------------------------------------------------|
| <b>Estática (early binding)</b> | En <b>tiempo de compilación</b> | Más rápida, menos flexible         | Variables globales, métodos <code>final</code> en Java                    |
| <b>Dinámica (late binding)</b>  | En <b>tiempo de ejecución</b>   | Más flexible, pero menos eficiente | Polimorfismo, <code>virtual</code> en C++, <code>dynamic</code> en Python |

## Ejemplo de Ligadura Estática

En lenguajes con tipado estático, como **C** o **Java**, el tipo de una variable o función se determina en **tiempo de compilación**, lo que permite optimizaciones por parte del compilador.

```

class Animal {
 void hacerSonido() {
 System.out.println("Sonido genérico");
 }
}

public class Main {
 public static void main(String[] args) {
 Animal a = new Animal();
 a.hacerSonido(); // La ligadura es estática, siempre ejecuta "Sonido
genérico"
 }
}

```

## Ejemplo de Ligadura Dinámica

En lenguajes con soporte para **polimorfismo y herencia**, como **Java, C++ o Python**, la ligadura se puede realizar en **tiempo de ejecución**, dependiendo del tipo real del objeto.

```

class Animal {
 void hacerSonido() {
 System.out.println("Sonido genérico");
 }
}

class Perro extends Animal {
 void hacerSonido() {
 System.out.println("Ladrido");
 }
}

public class Main {
 public static void main(String[] args) {
 Animal a = new Perro(); // Ligadura dinámica
 a.hacerSonido(); // Ejecuta "Ladrido" porque se resuelve en tiempo de
ejecución
 }
}

```

✓ En este caso, aunque `a` es del tipo `Animal`, en **tiempo de ejecución** se detecta que es una instancia de `Perro`, por lo que se llama al método sobreescrito.

# Diferencias Clave entre Ligadura Estática y Dinámica

| Característica            | Ligadura Estática                     | Ligadura Dinámica                       |
|---------------------------|---------------------------------------|-----------------------------------------|
| <b>Cuándo se resuelve</b> | En tiempo de <b>compilación</b>       | En tiempo de <b>ejecución</b>           |
| <b>Velocidad</b>          | Más rápida (optimización)             | Más lenta (búsqueda en tiempo real)     |
| <b>Flexibilidad</b>       | Menos flexible                        | Más flexible                            |
| <b>Uso común</b>          | Variables locales, funciones normales | Polimorfismo, sobreescritura de métodos |

## Conclusión:

- **Ligadura estática** mejora la eficiencia y evita errores en ejecución, pero limita la flexibilidad.
- **Ligadura dinámica** permite la reutilización de código con polimorfismo, aunque con un pequeño costo de rendimiento.