# Práctica Nro. 3 Semántica

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

**Ejercicio 1:** ¿Qué define la semántica?

**RTA {**

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:

1. ¿Qué significa compilar un programa?

**RTA {**

Proceso por el cual, un programa escrito en lenguaje de alto nivel, se traduce a lenguaje objeto (lenguaje de máquina) o a lenguaje intermedio (lenguaje ensamblador). Este proceso es llevado a cabo por el compilador.

Se traduce/compila antes de ejecución.

▪ Pasa por todas las instrucciones antes de la ejecución (ventajas y desventajas).

▪ El código que se genera se guarda y se puede reusar ya compilado.

▪ La compilación implica varias etapas.

**}**

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

**RTA {**

La compilación puede ejecutarse en una o dos etapas. En ambos casos se cumplen varias subetapas, las principales son:

**1) Etapa de Análisis:**

Análisis léxico (Programa Scanner)

Análisis sintáctico (Programa Parser)

Análisis semántico (Programa de Semántica estática)

*//🡪 Puede generarse código intermedio.*

**2) Etapa de Síntesis:**

Optimización del código

Generación del código final

**1) Más vinculado al código fuente.**

**2) Más vinculado a características del código objeto y del hardware y arquitectura**.

**}**

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

**RTA {**

**Es el tercer paso de la Etapa de Análisis.**

Análisis semántica (semántica estática):

* Debe pasar antes bien Scanner y Parser
* Fase medular, una de las más importantes
* Procesa las estructuras sintácticas (reconocidas por el analizador sintáctico)
* Agrega otro tipo de información implícita
* y la estructura del código ejecutable continúa tomando forma.

Realiza la comprobación de tipos (aplica

gramática de atributos)

* Agrega a la tabla de símbolos los descriptores de tipos
* Realiza comprobaciones de duplicados, problema de tipos, etc.
* Realiza comprobaciones de nombres. (toda variable debe estar declarada en su entorno, variables no declaradas)
* Es el nexo entre etapas inicial y final del compilador (Análisis y Síntesis)

**}**

**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.

**RTA {**

El Intérprete cuenta con una serie de herramientas para la traducción a lenguaje de máquina. Por cada posible acción hay un subprograma en lenguaje de máquina que ejecuta esa acción. La interpretación se realiza llamando a estos subprogramas en la secuencia adecuada hasta generar el resultado de la ejecución.

Un intérprete ejecuta repetidamente la siguiente secuencia de acciones:

* Obtiene la próxima sentencia.
* Determina la acción a ejecutar.
* Ejecuta la acción.

No es lo mismo compilar e interpretar. En la compilación se debe ejecutar el proceso de compilar previamente a ejecutar un programa, mientras que, en la interpretación, se ejecuta el proceso de interpretar sentencia por sentencia en tiempo de ejecución.

**Intérprete:**

 Se utiliza el intérprete en la ejecución.

 Ejecuta el programa línea por línea.

 Por donde pase dependerá de la acción del usuario, de la entrada de datos y/o de alguna decisión del programa.

 Siempre se debe tener el Programa Intérprete.

 El programa fuente será público (necesito ambos).

**Compilador:**

 Se utiliza el compilador antes de la ejecución.

 Produce un programa ejecutable equivalente en lenguaje objeto.

 El programa fuente no será público.

Comparación entre compilador e interprete:

**Por el orden de ejecución:  
Intérprete:**

* Sigue el orden lógico de ejecución (no necesariamente recorre todo el código).

**Compilador:**

* Sigue el orden físico de las sentencias (recorre todo).

**Por el tiempo consumido de ejecución:**

**Intérprete:**

* Por cada sentencia que pasa realiza el proceso de decodificación (lee, analiza y ejecuta) para determinar las operaciones y sus operandos. Es repetitivo.
* Si la sentencia está en un proceso iterativo (ej.: for/while), se realizará la tarea de decodificación tantas veces como sea requerido.
* La velocidad de proceso se puede ver afectada por esto.

**Compilador:**

* Pasa por todas las sentencias.
* No repite lazos.
* Traduce todo de una sola vez.
* Genera código objeto ya compilado.
* La velocidad de compilar dependerá del tamaño del código.

**Por la eficiencia posterior:**

**Intérprete:**

* Más lento en ejecución. Se repite el proceso cada vez que se ejecuta el mismo programa o pasa por las mismas instrucciones.
* Para ser ejecutado en otra máquina se necesita tener sí o sí el intérprete instalado y el programa fuente será público.

**Compilador:**

* Más rápido de ejecutar desde el punto de vista del hardware porque ya está en un lenguaje de más bajo nivel.
* Detectó más errores al pasar por todas las sentencias.
* Está listo para ser ejecutado. Ya compilado es más eficiente.
* Por ahí tardó más en compilar porque se verifica todo previamente. El programa fuente no será público.

**Por la detección de errores:**

**Intérprete:**

* Las sentencias del código fuente pueden ser relacionadas directamente con la sentencia en ejecución, entonces se puede ubicar dónde se produjo el error.
* Es más fácil detectarlos por donde pasa la ejecución.
* Es más fácil corregirlos.

**Compilador:**

* Se pierde la referencia entre el código fuente y el código objeto.
* Es casi imposible ubicar el error, pobres en significado para el programador.
* Se deben usar otras técnicas (ej.: Semántica Dinámica).

**}**

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

**RTA {**

El error sintactico proviene de no obedecer las reglas de la gramática definidas para un lenguaje de programación. Mientras que el error semántico se da cuando, no hay error sintactico, es decir, la sentencia es “valida”. Pero, se viola alguna regla semántica del lenguaje. Por ejemplo:

En Java:

“ int i := 5” 🡨 Error sintactico, la asignación se hace con “=”.

“ int i = 5;

String chain = “Una cadena de texto”;

i = i + chain; 🡨 Error semantico, no esta definida la suma entre un entero y un String en Java.

RESPUESTA DE CHATGPT:

### **1. Error Sintáctico**

Un error sintáctico ocurre cuando el código no sigue las reglas gramaticales del lenguaje de programación. Es similar a escribir una oración con mala gramática en un idioma. Estos errores son detectados por el compilador o intérprete antes de ejecutar el programa.

#### **Ejemplo de error sintáctico (en Python):**

python

print "Hola, mundo"

🔴 **Error:** Falta el uso de paréntesis en print en Python 3.  
✅ **Corrección:**

python

print("Hola, mundo")

Otro ejemplo:

python

if (x > 10

print("Mayor que 10")

🔴 **Error:** Falta el paréntesis de cierre en la condición del if.

* Un **error semántico** ocurre cuando se usan elementos del lenguaje de manera incorrecta según su significado, incluso si la sintaxis es válida. Un ejemplo típico es intentar sumar un número con una cadena en Python.
* Un **error lógico** ocurre cuando el programa se ejecuta sin errores, pero no produce el resultado esperado debido a fallos en la lógica del código.

### **Ejemplo de error semántico en Python**

python

x = 10

y = "5"

resultado = x + y # Error: no se puede sumar un entero con una cadena

🔴 **Error:**

bash

TypeError: unsupported operand type(s) for +: 'int' and 'str'

**}**

**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.*

1. Pascal Program P ***var*** ***5***: integer;

***var*** a: char; Begin

for i:=5 to 10 do begin write(a); a=a+1;

end;

End.

* Falta un “;” luego de la sentencia “Program P”. Error sintáctico, detectado en compilación.
* La palabra clave “var” solo se escribe una única vez. Error semántico, detectado en compilación.
* El ID de una variable no puede comenzar con un número. Error sintáctico, detectado en compilación.
* La variable “a” jamás fue inicializada. Error semántico, detectado en ejecución.
* No se puede sumar una variable de tipo String con un integer. Error semántico, detectado en ejecución.

*Ayuda: Sintáctico 2, Semántico 3*

1. 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*

* En la declaración de parámetros, va “ArrayList” en vez de “arrayList”. Error sintáctico, detectado en compilación.
* En el for, falta la palabra reservada “int” antes de “i”. Error semántico, detectado en compilación.
* El for entrara en un bucle infinito. Error lógico, detectado en ejecución.
* En “listado.get(listado.size()-1)=(BOOLEAN) numero>i;” la forma correcta de hacer una asignación de valor en un listado, es con .set y no con .get. Y no se usa la asignación “=”. Error sintáctico, detectado en compilación.
* No es correcto (BOOLEAN) sino que debería ir (Boolean). Error sintáctico, detectado en compilación.
* No es correcto iniciar un String vacío usando “null”, debería usarse “”. Error lógico, detectado en ejecución.
* Se retorna un valor booleano, cuando debería retornarse un String. Error semántico, detectado en compilación.

1. 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*

1. 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)--->> ") Ayuda: *Sintácticos 2, Semánticos 3*

* “while r = true:” La comparación se hace con “==”, error sintáctico, detectado en compilación.
* No tiene sentido la sentencia “while i <= (N^0.5)” en el contexto del propósito del programa. Error lógico, detectado luego de la ejecución.
* X3: No hay paréntesis en el primer, segundo y tercer print. Error sintáctico, detectado en compilación.

1. 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*

**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.*

**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?

**RTA {**

## **1.** self **en Ruby**

📌 **¿Qué es self?**

* self es una **referencia al objeto actual** dentro de un contexto en Ruby.
* Su valor depende del lugar donde se invoque: puede ser una instancia de una clase, una clase en sí misma, o incluso un módulo.
* Se usa para llamar métodos de instancia, definir atributos, y distinguir variables de instancia de las locales.

📌 **Ejemplo de self en una clase:**

ruby

CopiarEditar

class Ejemplo

def initialize(nombre)

@nombre = nombre

end

def mostrar

puts self # Se refiere a la instancia actual del objeto

end

end

obj = Ejemplo.new("Ruby")

obj.mostrar # Imprime: #<Ejemplo:0x000055b9d2c5fbd0>

🔹 En este caso, self es la instancia del objeto (obj).

📌 **Ejemplo de self en métodos de clase:**

ruby

CopiarEditar

class Ejemplo

def self.metodo\_de\_clase

puts "Este es un método de clase"

end

end

Ejemplo.metodo\_de\_clase # Llamada sin necesidad de instanciar

🔹 Aquí, self hace referencia a la **clase misma (Ejemplo)**, permitiendo definir un método de clase.

## **2.** nil **en Ruby**

📌 **¿Qué es nil?**

* nil representa **la ausencia de un valor o un objeto inexistente** en Ruby.
* Equivalente a null en otros lenguajes como Java o Python.
* Es un objeto de la clase NilClass.

📌 **Ejemplo de nil:**

ruby

CopiarEditar

variable = nil

puts variable.nil? # Imprime: true

🔹 nil? es un método que verifica si una variable es nil.

📌 **Uso común de nil:**

* Retorno por defecto cuando un método no devuelve nada:

ruby

CopiarEditar

def metodo\_sin\_retorno

end

puts metodo\_sin\_retorno.nil? # Imprime: true

* Para indicar valores opcionales o inexistentes en estructuras de datos:

ruby

CopiarEditar

hash = { nombre: "Juan", edad: nil }

puts hash[:edad] # Imprime: nada (nil)

📌 **Cuidado con nil en operaciones:**

ruby

CopiarEditar

puts nil + 5 # ❌ Error: NoMethodError (undefined method `+' for nil:NilClass)

🔹 No se puede operar con nil directamente sin manejarlo antes.

**}**

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

**RTA {**

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

## **🔹** null **en JavaScript**

📌 **¿Qué es null?**

* null es un valor **intencionalmente vacío** o **sin valor** asignado manualmente por el programador.
* Es un **tipo de dato** en JavaScript (typeof null devuelve "object", aunque esto es un error histórico del lenguaje).
* Se usa cuando queremos indicar que una variable **no tiene valor, pero debería tenerlo en el futuro**.

## **🔹** undefined **en JavaScript**

📌 **¿Qué es undefined?**

* undefined significa que una variable ha sido **declarada pero no tiene un valor asignado**.
* También es el valor por defecto que devuelve una función sin return.
* Es un **tipo primitivo** en JavaScript.

🔹 Una función que no tiene return devuelve undefined por defecto.

## **🔹 Comparaciones entre** null **y** undefined

📌 **Comparación con == (igualdad no estricta)**

javascript

CopiarEditar

console.log(null == undefined); // true

🔹 Ambos son considerados "similares" en comparaciones no estrictas.

📌 **Comparación con === (igualdad estricta)**

javascript

CopiarEditar

console.log(null === undefined); // false

🔹 Son diferentes tipos de datos (null es un "object", undefined es "undefined").

**}**

**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

**RTA {**

La sentencia **break** se utiliza en varios lenguajes de programación para **salir abruptamente** de estructuras de control como bucles (for, while, do-while) y, en algunos casos, estructuras switch. A continuación, se explica su comportamiento en **C, PHP, JavaScript y Ruby**.

## **1️⃣** break **en C**

📌 **Características principales:**

* Se usa en **bucles** (for, while, do-while) y en la estructura switch-case.
* Detiene **inmediatamente** la ejecución del bucle o switch, continuando con el código posterior.
* No puede usarse fuera de un bucle o switch (generará error de compilación).

📌 **Ejemplo en un bucle for:**

c

CopiarEditar

#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 ", i);

}

return 0;

}

🔹 **Salida:** 0 1 2 3 4 (el bucle se detiene en 5).

📌 **Ejemplo en switch-case:**

c

CopiarEditar

#include <stdio.h>

int main() {

int opcion = 2;

switch (opcion) {

case 1:

printf("Opción 1\n");

break;

case 2:

printf("Opción 2\n");

break; // Evita la ejecución de otros casos

case 3:

printf("Opción 3\n");

}

return 0;

}

🔹 **Salida:** Opción 2 (sin break, ejecutaría también Opción 3).

## **2️⃣** break **en PHP**

📌 **Características principales:**

* Se usa en **bucles** (for, while, do-while, foreach) y en switch-case.
* Puede recibir un **número opcional** (break n;), indicando cuántos niveles de bucle debe romper.
* Permite salir de múltiples bucles anidados de una vez.

📌 **Ejemplo básico en while:**

php

CopiarEditar

<?php

$i = 1;

while ($i < 10) {

if ($i == 5) {

break; // Rompe el bucle cuando i es 5

}

echo $i . " ";

$i++;

}

?>

🔹 **Salida:** 1 2 3 4

📌 **Ejemplo con break 2; (salida de 2 niveles de bucles):**

php

CopiarEditar

<?php

for ($i = 1; $i <= 3; $i++) {

for ($j = 1; $j <= 3; $j++) {

if ($j == 2) {

break 2; // Rompe los dos bucles

}

echo "i=$i, j=$j\n";

}

}

?>

🔹 **Salida:** i=1, j=1 (rompe ambos bucles cuando j == 2).

## **3️⃣** break **en JavaScript**

📌 **Características principales:**

* Se usa en **bucles** (for, while, do-while) y en switch-case.
* No acepta valores opcionales como en PHP.
* Puede usarse en **bucles etiquetados**, permitiendo romper múltiples niveles de anidación.

📌 **Ejemplo en for:**

javascript

CopiarEditar

for (let i = 0; i < 10; i++) {

if (i === 5) {

break; // Sale del bucle cuando i es 5

}

console.log(i);

}

🔹 **Salida:** 0 1 2 3 4

📌 **Ejemplo con switch-case:**

javascript

CopiarEditar

let opcion = 2;

switch (opcion) {

case 1:

console.log("Opción 1");

break;

case 2:

console.log("Opción 2");

break;

case 3:

console.log("Opción 3");

}

🔹 **Salida:** Opción 2 (sin break, ejecutaría también Opción 3).

📌 **Ejemplo con bucles etiquetados (label):**

javascript

CopiarEditar

outerLoop: for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

if (j == 1) {

break outerLoop; // Rompe ambos bucles

}

console.log(`i=${i}, j=${j}`);

}

}

🔹 **Salida:** i=0, j=0 (rompe los dos bucles cuando j == 1).

## **4️⃣** break **en Ruby**

📌 **Características principales:**

* Se usa en **bucles** (while, for, until, each, loop).
* También puede usarse dentro de bloques (each, times, upto).
* Rompe **solo el bucle más interno** y continúa con el código siguiente.

📌 **Ejemplo en while:**

ruby

CopiarEditar

i = 0

while i < 10

if i == 5

break # Sale del bucle cuando i es 5

end

print "#{i} "

i += 1

end

🔹 **Salida:** 0 1 2 3 4

📌 **Ejemplo en each:**

ruby

CopiarEditar

[1, 2, 3, 4, 5].each do |num|

break if num == 3 # Sale del bucle cuando num es 3

puts num

end

🔹 **Salida:** 1 2

📌 **Ejemplo en loop (bucle infinito):**

ruby

CopiarEditar

loop do

numero = rand(1..10)

puts numero

break if numero == 7 # Rompe el bucle cuando el número aleatorio es 7

end

🔹 **Salida:** Detenida cuando se genera 7.

## **🔹 Comparación entre los lenguajes**

| **Lenguaje** | **Uso en bucles** | **Uso en switch-case** | **Permite romper múltiples bucles** | **Soporta etiquetas (label)** |
| --- | --- | --- | --- | --- |
| **C** | ✅ Sí | ✅ Sí | ❌ No | ❌ No |
| **PHP** | ✅ Sí | ✅ Sí | ✅ Sí (break n;) | ❌ No |
| **JavaScript** | ✅ Sí | ✅ Sí | ✅ Sí (con label) | ✅ Sí (label:) |
| **Ruby** | ✅ Sí | ❌ No (no tiene switch) | ❌ No | ❌ No |

## **🔹 Conclusión**

* **En todos los lenguajes, break se usa para salir de bucles.**
* **C, PHP y JavaScript permiten break en switch-case, Ruby no.**
* **PHP permite romper múltiples niveles de bucles con break n;.**
* **JavaScript permite break en bucles etiquetados (label:).**

**}**

# 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)

**RTA {**

Hay que asociar cada entidad a sus atributos. Es un concepto central en la definición de la semántica de los lenguajes de programación Por ejemplo:

int a;

Entidad: Variable Atributos:

Nombre: a

Tipo: int entero

Rango de valores y operaciones: determinado por el tipo

**Los programas trabajan con entidades**

**Las entidades tienen atributos**

**Estos atributos tienen que establecerse (tener un valor) antes de poder usar la entidad**

**LIGADURA: es el momento en el que el atributo se asocia con un valor determinado**

**Ligadura es Estática:**

1. Se establece antes de la ejecución.

2. No se puede modificar.

El termino estática referencia al binding time y a su estabilidad.

**Ligadura es Dinámica:**

1. Se establece durante la ejecución

2. Si puede modificarse durante ejecución de acuerdo

a alguna regla especifica del lenguaje.

* Excepción: constantes (el binding es en ejecución/runtime pero no puede ser modificado luego de establecido)

RESPUESTA DE CHATGPT:

## **Definición de Ligadura y su Importancia en la Semántica de un Programa**

### **🔹 ¿Qué es la Ligadura?**

La **ligadura (binding)** es el proceso mediante el cual se asocia un **nombre** con una **entidad** en un programa, como una variable, función, tipo de dato o método.

📌 **Ejemplo:**

python

CopiarEditar

x = 10 # Se establece una ligadura entre 'x' y el valor 10

Aquí, x está ligado al valor 10.

### **🔹 Importancia de la Ligadura en la Semántica de un Programa**

* Define **cómo y cuándo** se establecen las asociaciones en el programa.
* Determina **el ámbito** y **la visibilidad** de variables y funciones.
* Afecta la **resolución de nombres**, lo que impacta en **el rendimiento y comportamiento** del código.

## **📌 Tipos de Ligadura: Estática vs. Dinámica**

| **Característica** | **Ligadura Estática (Early Binding)** | **Ligadura Dinámica (Late Binding)** |
| --- | --- | --- |
| **Cuándo se resuelve** | En **tiempo de compilación** | En **tiempo de ejecución** |
| **Velocidad** | Más rápida (optimización del compilador) | Más lenta (requiere más procesamiento) |
| **Flexibilidad** | Menos flexible, pero más segura | Más flexible, pero propensa a errores |
| **Ejemplo de uso** | Variables con **tipos fijos** | Métodos sobrecargados, polimorfismo en POO |

## **📌 Ejemplos de Ligadura Estática vs. Dinámica**

### **1️⃣ Ligadura Estática (Early Binding)**

Se resuelve **antes de la ejecución**, en tiempo de **compilación**.

📌 **Ejemplo en C (tipado estático)**

c

CopiarEditar

int x = 5; // La variable 'x' está ligada a un entero en tiempo de compilación

Aquí, el tipo y el valor de x se determinan **antes de la ejecución**.

📌 **Ejemplo en Python (función con tipado estático - opcional con anotaciones)**

python

CopiarEditar

def sumar(a: int, b: int) -> int:

return a + b

Aunque Python es dinámico, las anotaciones de tipo pueden indicar una **ligadura estática**.

### **2️⃣ Ligadura Dinámica (Late Binding)**

Se resuelve **durante la ejecución**, lo que permite más flexibilidad pero también más sobrecarga de procesamiento.

📌 **Ejemplo en Python (tipado dinámico)**

python

CopiarEditar

x = 5 # 'x' es un entero ahora

x = "Hola" # Ahora 'x' es un string (la ligadura cambia en ejecución)

🔹 x cambia su tipo en tiempo de ejecución, mostrando una **ligadura dinámica**.

📌 **Ejemplo en JavaScript (Polimorfismo - Ligadura Dinámica)**

javascript

CopiarEditar

class Animal {

hacerSonido() {

console.log("Sonido genérico");

}

}

class Perro extends Animal {

hacerSonido() {

console.log("Guau!");

}

}

let mascota = new Perro();

mascota.hacerSonido(); // En ejecución, se resuelve a "Guau!"

🔹 JavaScript usa **ligadura dinámica** para determinar qué método llamar en tiempo de ejecución.

## **📌 Conclusión**

* **Ligadura Estática:** Más rápida y segura, pero menos flexible (C, Java con tipado fuerte).
* **Ligadura Dinámica:** Más flexible, pero con mayor sobrecarga en ejecución (Python, JavaScript, Ruby).

📌 **¿Cuál usar?**

* Para **rendimiento y seguridad**, preferir **ligadura estática**.
* Para **flexibilidad y polimorfismo**, usar **ligadura dinámica**.

**}**