
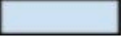
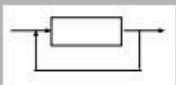
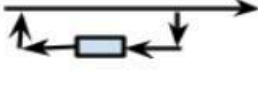



Práctica 2 - Sintaxis

1)

Meta símbolos utilizados por		Símbolo utilizado en Diagramas sintacticos	Significado
BNF	EBNF		
palabra terminal	palabra terminal		Definición de un elemento terminal
< >	< >	rectángulo 	Definición de un elemento no terminal
::=	::=	diagrama con rectángulos, óvalos y flechas	Definición de una producción
	()	flecha que se divide en dos o más caminos	Selección de una alternativa
< p > < p1 >	{ }		Repetición
	*		Repetición de 0 o más veces
	+		Repetición de 1 o más veces
	[]		Opcional, está presente o no lo está

2) La sintaxis establece reglas que definen cómo deben combinarse las componentes básicas, llamadas "word", para formar sentencias y programas. Elementos:

- Alfabeto o conjuntos de caracteres.
- Identificadores.
- Operadores.
- Palabra clave y palabra reservada.
- Comentarios y uso de blancos.

3)

Reglas lexicográficas: Se refiere al conjunto de normas que rigen la formación de los tokens en un lenguaje de programación. Estas reglas definen cómo se identifican palabras clave, identificadores, operadores y otros elementos básicos del lenguaje.

- Se aplican en la fase de análisis léxico de un compilador o intérprete.
- Un token es la unidad mínima significativa del código fuente, como una palabra clave (`if`, `while`), un operador (`+`, `-`, `*`), un número (`42`), etc.
- Ejemplo de regla lexicográfica: En Python, un identificador sólo puede comenzar con una letra o un guión bajo, pero no con un número.

Regla Sintáctica: Se refiere a las normas que dictan cómo los **tokens** pueden combinarse para formar expresiones, sentencias y estructuras válidas en un lenguaje.

- Ejemplo de regla sintáctica: En C, una estructura `if` debe estar escrita en la forma `if (condición) { bloque de código; }`.

4) **Palabra reservada:** Las palabras reservadas son identificadores predefinidos en un lenguaje de programación que tienen un significado especial y no pueden usarse como nombres de variables, funciones u otros identificadores definidos por el usuario. Son parte de la sintaxis del lenguaje y su uso está restringido.

- Permiten al compilador y al programador expresarse claramente.
- Hacen los programas más legibles y permiten una rápida traducción.

En una **gramática formal**, las palabras reservadas son equivalentes a los **símbolos terminales**, ya que forman parte del lenguaje y no pueden ser reemplazadas por otras estructuras.

5)a) La gramática está definida como $G = (N, T, S, P)$, donde:

Símbolos no terminales (N):

- **<numero_entero>**: Representa un número entero.
- **<dígito>**: Representa un solo dígito del 0 al 9.

Símbolos terminales (T):

- $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (los dígitos del sistema decimal).

Símbolo inicial (S)

- `<numero_entero>` (es el punto de inicio para derivar cadenas válidas en el lenguaje)

Reglas de producción (P):

`<numero_entero> ::= <digito><numero_entero> | <numero_entero><digito> | <digito>`
`<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

Estas reglas indican que un número entero puede ser un solo dígito (`<digito>`) o una combinación de varios dígitos, con la posibilidad de agregarlos antes o después de otro número entero.

5)b) Una gramática es ambigua si una sentencia puede derivarse de más de una forma. Esto significa que la gramática permite múltiples estructuras para la misma cadena, lo que hace que sea ambigua.

Si queremos generar la cadena "23", podemos hacerlo de dos maneras:

1. Usando la producción `<numero_entero> ::= <digito> <numero_entero>`
2. Usando la producción `<numero_entero> ::= <numero_entero> <digito>`

Ambas derivaciones producen la misma cadena, pero con diferentes estructuras, lo que hace que la gramática sea ambigua.

Corrección de la ambigüedad

Para evitar la ambigüedad, podemos definir la gramática de manera **recursiva por la izquierda** o **por la derecha**, pero no ambas. Una versión corregida sería:

`<numero_entero> ::= <digito> | <numero_entero> <digito>`
`<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

- Un **dígito** es un **terminal** (valor atómico, no se descompone más).
- Un **número entero** es una **secuencia de dígitos**, construida recursivamente usando la regla `<numero_entero> ::= <numero_entero> <digito>`.

6) Para definir una **palabra cualquiera** en **BNF (Backus-Naur Form)**, debemos considerar que una palabra es una secuencia de letras.

Si asumimos que una palabra está compuesta solo por letras minúsculas y mayúsculas del alfabeto inglés (a-z y A-Z), podemos definir la gramática así:

```
<palabra> ::= <letra> | <palabra> <letra>
<letra> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
           "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |
           "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
           "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
```

<palabra> representa una palabra, que puede ser:

1. Un solo **<letra>**, lo que permite palabras de un solo carácter.
2. Una **<palabra>** seguida de otra **<letra>**, lo que permite palabras de múltiples caracteres.

<letra> define los caracteres permitidos (las letras del alfabeto).

7) En BNF (Backus-Naur Form), debemos definir un número real como una secuencia de dígitos que puede contener un punto decimal y, opcionalmente, un signo (+ o -).

```
<numero_real> ::= <signo> <parte_entera> <parte_decimal> | <parte_entera>
<parte_decimal>
<signo> ::= "+" | "-"
<parte_entera> ::= <dígito> | <parte_entera> <dígito>
<parte_decimal> ::= "." <dígito> <digitos_opcionales>
<dígito> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<digitos_opcionales> ::= "" | <dígito> <digitos_opcionales>
```

<numero_real>

- Puede comenzar con un signo (+ o -), pero no es obligatorio.
- Debe tener una parte entera.
- Puede o no contener una parte decimal.

<parte_entera>

- Es una secuencia de dígitos (al menos un dígito obligatorio).

<parte_decimal>

- Puede ser vacía (""), lo que permite representar números enteros.
- Si existe, comienza con un punto (.) seguido de al menos un dígito
- Puede tener más dígitos opcionales (<digitos_opcionales>), lo que permite escribir decimales de cualquier longitud.

Gramática en EBNF para números reales

La **EBNF (Extended Backus-Naur Form)** permite simplificar la gramática usando:

- [...] para elementos opcionales.
- {...} para repeticiones de 0 o más veces
- (...) para agrupar elementos.

<numero_real> ::= [<signo>] <parte_entera> [<parte_decimal>]

<signo> ::= "+" | "-"

<parte_entera> ::= <digito> {<digito>}*

<parte_decimal> ::= "." <digito> {<digito>}*

<digito> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Característica	BNF	EBNF
Expresividad	Más rígida, requiere más reglas.	Más flexible y concisa.
Repetición	Se logra con reglas recursivas.	Usa {...} para definir repeticiones.
Opcionalidad	Se maneja con reglas adicionales ("").	Usa [...] para elementos opcionales.
Legibilidad	Puede volverse extensa y difícil de leer.	Más compacta y fácil de interpretar.

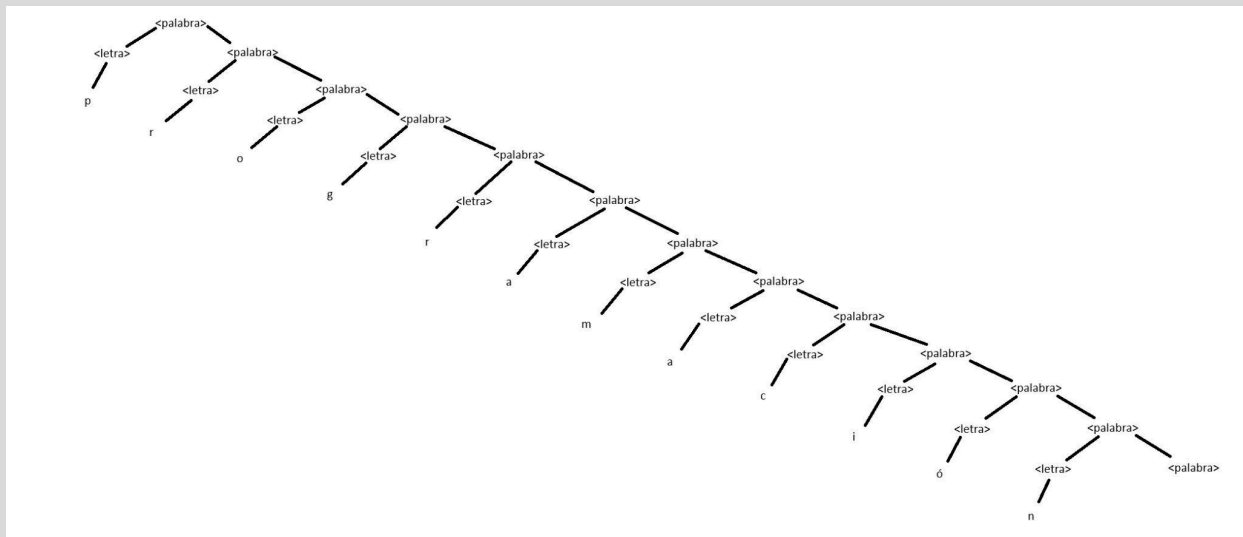
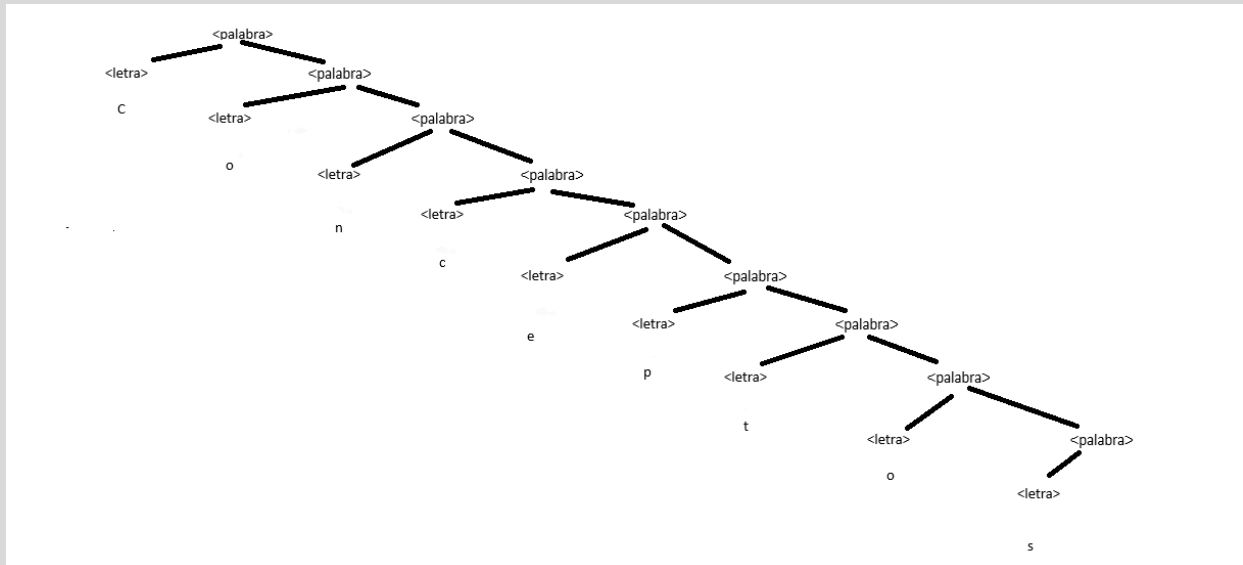
8)

<oracion> ::= <palabra> | <palabra> <espacio> <oracion>

<palabra> ::= <letra> | <letra> <palabra>

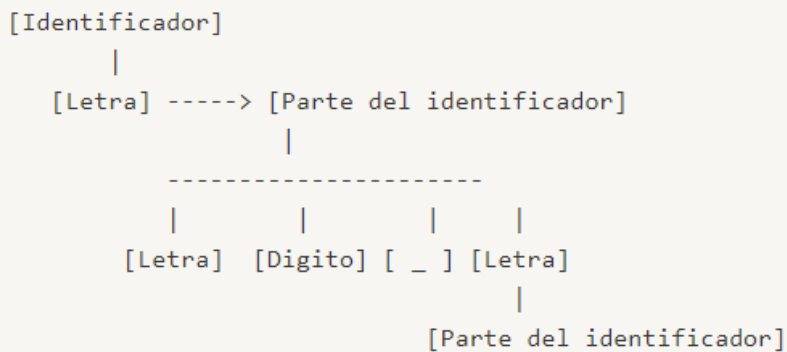
<letra> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |
"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
"N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |

"á" | "é" | "í" | "ó" | "ú" | "Á" | "É" | "Í" | "Ó" | "Ú"
 <espacio> ::= " "



<numero_real> ::= <signo> <parte_entera> <parte_decimal> | <parte_entera>
 <parte_decimal>
 <signo> ::= "+" | "-"
 <parte_entera> ::= <dígito> | <dígito> <parte_entera>
 <parte_decimal> ::= "." <dígito> <digitos_opcionales>
 <dígito> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
 <digitos_opcionales> ::= "" | <dígito> <digitos_opcionales>

- **<letra>**: Son las letras del alfabeto en minúsculas y mayúsculas (A-Z, a-z).
- **<dígito>**: Son los números del 0 al 9.
- **<cuerpo_identificador>**: Después de la primera letra o guión bajo, el cuerpo de un identificador puede contener letras o dígitos (pero no puede comenzar con un dígito).
- **ε**: Representa la opción vacía, lo que significa que el cuerpo del identificador puede terminar después de la primera letra o guión bajo.



10) Defina con EBNF la gramática para una expresión numérica, donde intervienen variables y números. Considere los operadores +, -, * y / sin orden de prioridad. No considere el uso de paréntesis.

Sin prioridad:

```

P = {
  <expresión> ::= ( <numero> | <variable> ) { [ <operador> ] ( <numero> | <variable> ) }*
  <número> ::= [ <signo> ] { <dígito> }+
  <variable> ::= <letra> [ ( <letra> | <dígito> ) ]*
  <operador> ::= "+", "-", "*", "/"

  <signo> ::= "+", "-"
  <dígito> ::= "0".."9"
  <letra> ::= "a".."z"

}
  
```

Con prioridad:

$G=(N,T,S,P)$

$N = \{ \langle \text{expresión} \rangle, \langle \text{termino} \rangle, \langle \text{número} \rangle, \langle \text{variable} \rangle, \langle \text{operador} \rangle, \langle \text{signo} \rangle, \langle \text{dígito} \rangle, \langle \text{letra} \rangle \}$

$T = \{ "+", "-", "*", "/", "0".."1", "a".."z", "A".."Z" \}$

$S = \langle \text{expresión} \rangle$

- El símbolo inicial es el no terminal que se usa como punto de partida para derivar una cadena de símbolos. En esta gramática, el símbolo inicial es $\langle \text{expresión} \rangle$, ya que es la primera regla de producción que aparece.

$P = \{$

$\langle \text{expresión} \rangle ::= \langle \text{termino} \rangle \{ ("+" | "-") \langle \text{termino} \rangle \}^*$

$\langle \text{termino} \rangle ::= (\langle \text{número} \rangle | \langle \text{variable} \rangle) \{ ("0*" | "/") \langle \text{termino} \rangle \}^*$

$\langle \text{número} \rangle ::= [\langle \text{signo} \rangle] \{ \langle \text{dígito} \rangle \}^+$

$\langle \text{variable} \rangle ::= \langle \text{letra} \rangle [(\langle \text{letra} \rangle | \langle \text{dígito} \rangle)]^*$

$\langle \text{operador} \rangle ::= "+", "-", "*", "/"$

$\langle \text{signo} \rangle ::= "+", "-"$

$\langle \text{dígito} \rangle ::= "0".."1"$

$\langle \text{letra} \rangle ::= "a".."z"$

$\}$

11)

- **Definición de $\langle \text{variable} \rangle$:** Debería permitir letras seguidas de dígitos.
- **Uso de IN en $\langle \text{sentencia_for} \rangle$:** Cambiar IN por una estructura de rango o contador más común.
- **Definición de $\langle \text{bloque} \rangle$:** Simplificar la regla para evitar ambigüedades.
- **Definición de $\langle \text{cadena} \rangle$:** Asegurarse de que $\langle \text{otro} \rangle$ esté bien definido o eliminarlo.
- **Regla $\langle \text{sentencia} \rangle$:** No es necesariamente incorrecta, pero asegúrate de que las sentencias estén bien definidas en el contexto.
- **Uso de $\langle \text{operacion} \rangle$:** Si no se utiliza, eliminarla de la gramática.
- **Expresiones matemáticas:** Si se requiere, agregar una regla para las operaciones.

Práctica 3 - Semántica

1) Describe el significado de los símbolos, palabras y frases de un lenguaje ya sea lenguaje natural o lenguaje informático. Es decir, describe qué hacen las instrucciones y expresiones cuando se ejecutan.

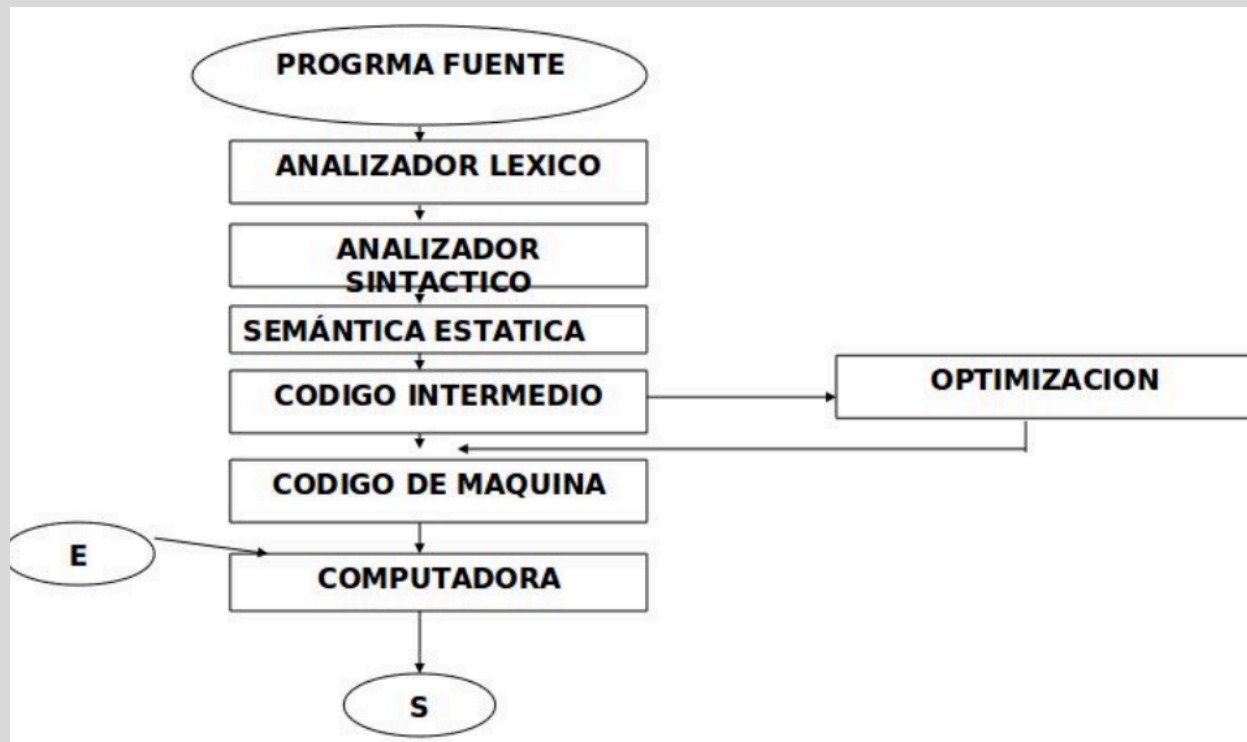
Es fundamental para entender **cómo se ejecutan los programas, qué resultados producen y cómo razonar sobre su comportamiento.**

2)a) Compilar un programa significa traducir el código fuente escrito en un lenguaje de alto nivel (como C, Java o Python) a un código en lenguaje de máquina o intermedio que pueda ser ejecutado por la computadora. Este proceso lo realiza un **compilador** y permite que el programa sea ejecutado eficientemente.

Proceso de compilación:

- **Compilados:** C, C++, Rust (se traducen completamente antes de ejecutarse).
- **Interpretados:** Python, JavaScript (se traducen línea por línea en tiempo de ejecución).
- **Híbridos:** Java, Dart (se compilan a bytecode y luego se interpretan o se optimizan en tiempo de ejecución).

2)b)



Análisis léxico:

- Se divide el código en tokens (palabras clave, identificadores, operadores, etc.).
- Ejemplo: En `int x = 10;`, se identifican los tokens `int`, `x`, `=`, `10`, `;`.

Análisis sintáctico (Parsing):

- Verifica que la estructura del código sea válida según la gramática del lenguaje.
- Usa árboles de sintaxis abstracta (AST)
- Ejemplo: Detecta si falta un `;` en `int x = 10`.

Análisis semántico:

- Comprueba el significado del código, asegurando que las operaciones sean válidas.
- Las reglas de tipado estático en lenguajes como Java ayudan a evitar errores en el tiempo de ejecución.
- Ejemplo: En `int x = "hello";`, detecta un error porque `"hello"` no es un entero.

Generación de código intermedio:

- Transforma el código fuente en una representación intermedia (como bytecode en Java).
- Esto permite optimizaciones y compatibilidad entre arquitecturas.

Optimización de código:

- Mejora el código intermedio eliminando redundancias y optimizando instrucciones.
- Ejemplo: Convertir `x = x + 0;` en `x;` (eliminando una operación innecesaria).

Generación de código máquina:

- Traduce el código optimizado a instrucciones de bajo nivel que la CPU puede ejecutar.

Enlazado (Linking):

- Combina el código compilado con bibliotecas y otros módulos necesarios para la ejecución.
- Puede ser estático (cuando todo se une en un solo binario) o dinámico (cuando usa bibliotecas compartidas en tiempo de ejecución).

2)c) La semántica interviene en el **análisis semántico**, que es una de las fases de la compilación. Su importancia dentro de la compilación radica en garantizar que las operaciones realizadas en el código tengan sentido y cumplan con las reglas del lenguaje.

Por ejemplo, verifica que las variables están declaradas antes de usarlas, que los tipos de datos sean compatibles en las operaciones y que no haya errores como intentar dividir entre cero.

Si se detectan errores semánticos, el compilador genera mensajes de error para evitar que el programa continúe con una lógica incorrecta. Sin esta fase, un programa podría compilarse pero fallar en tiempo de ejecución por errores lógicos o de tipo.

3) **Compilar** un programa implica traducir todo el código antes de ejecutarlo, lo que mejora la velocidad y eficiencia. **Interpretarlo** significa traducirlo sobre la marcha (línea por línea), lo que da flexibilidad pero reduce el rendimiento. Algunos lenguajes como Java combinan ambas técnicas para aprovechar lo mejor de cada una.

Característica	Compilación	Interpretación
Velocidad de ejecución	Alta, ya que el código está optimizado y traducido previamente.	Baja, ya que el código se traduce en tiempo de ejecución.
Portabilidad	Baja, el código compilado suele ser específico de una arquitectura.	Alta, el código fuente se ejecuta en cualquier máquina con el intérprete adecuado.
Errores	Detecta errores antes de la ejecución.	Puede ejecutarse hasta encontrar un error.
Tiempo de inicio	Mayor, ya que la compilación requiere procesamiento previo.	Menor, ya que el código comienza a ejecutarse de inmediato.
Ejemplo de lenguajes	C, C++, Rust.	Python, JavaScript, Ruby.

4) Un error **sintáctico** está relacionado con un código que no sigue las reglas de gramática del lenguaje, mientras que un error **semántico** ocurre cuando la estructura del código es válida, pero su significado no tiene sentido dentro del contexto del programa.

Error sintáctico:

```
if (x == 10 # Falta el cierre del paréntesis
    print("x es 10")
```

Error semántico:

```
total = "100" + 20 # Intento sumar una cadena con un número
print(total)
```

<https://detailed-cathedral-9a9.notion.site/Pr-ctica-3-Ej5-1cc8804632d5809aacfffb1c81c81160>

5)a) Todos los errores, sintácticos o semánticos, ocurridos en Pascal ocurren en momento de compilación. Ya que es un lenguaje compilado.

a) Pascal

Program P ; # Error sintactico

var 5: integer; # variable arranca con numero => Error sintáctico

var a:char; # var se pone una unica vez => Error ????

Begin

for i:=5 to 10 do begin # Variable i no declarada => Error semantico

write(a);

a=a+1; Asignación con := => Error Sintactico

end; Suma de un char => Error semantico

End.

Ayuda: Sintáctico 2, Semántico 3

5)b)

b) Java:

public String tabla(int numero, arrayList<Boolean> listado) # ArrayList es el tipo => Error Sintactico

{

String result = null; # result = "" => Error Sintactico

for(i = 1; i < 11; i--) { # Loop infintio => Error Lógico # Variable i no declarada => Error semantico

result += numero + "x" + i + "=" + (i*numero) + "\n";

listado.get(listado.size()-1)=(BOOLEAN) numero>i; # (BOOLEAN) deberia ser Boolean => Error sintactico

} # ?? => Error semantico

return true; # Debe retornar String => Error semantico

}

Ayuda:

Sintácticos 4, Semánticos 3, Lógico 1

6) Ruby: **Self** representa el objeto actual en el contexto en el que se está ejecutando el código. Uso:

- Dentro de una clase, **self** hace referencia a la instancia del objeto.
- En un método de clase, **self** hace referencia a la propia clase.
- En el contexto global, **self** representa el objeto principal (**main**)

nil es un objeto especial en Ruby que representa la ausencia de valor o “nada”. Uso:

- Se usa cuando una variable no tiene valor asignado.
- Se devuelve en métodos cuando no hay un retorno explícito.

7)

Undefined: Indica que una variable ha sido declarada pero no le asignaron un valor. Es el valor predeterminado de las funciones que no retornan nada.

```
let a;  
console.log(a); // 🖱 undefined (variable declarada pero sin valor)  
  
function saludo() {}  
console.log(saludo()); // 🖱 undefined (la función no retorna nada)  
  
let persona = {};  
console.log(persona.nombre); // 🖱 undefined (propiedad inexistente)
```

Null: Representa ausencia intencional de valor.

```
let b = null;  
console.log(b); // 🖱 null (valor intencionalmente vacío)  
  
let persona = { nombre: "Juan", edad: 25 };  
persona = null; // 🖱 La referencia al objeto se elimina  
console.log(persona); // 🖱 null
```

8) La sentencia **break** se usa para salir de un bucle o estructura de control en diferentes lenguajes.

Lenguaje	Usado en	Soporte de múltiples niveles	Soporte de etiquetas	Retorno de valores
C	<code>for</code> , <code>while</code> , <code>do-while</code> , <code>switch</code>	✗ No	✗ No	✗ No
PHP	<code>for</code> , <code>while</code> , <code>do-while</code> , <code>foreach</code> , <code>switch</code>	✓ Sí (<code>break n;</code>)	✗ No	✗ No
JavaScript	<code>for</code> , <code>while</code> , <code>do-while</code> , <code>switch</code>	✗ No	✓ Sí (con <code>label:</code>)	✗ No
Ruby	<code>for</code> , <code>while</code> , <code>until</code> , <code>each</code>	✗ No	✗ No	✓ Sí (<code>break valor</code>)

9) La **ligadura** es la asociación entre un identificador (como una variable o función) y su valor o significado en un programa. Es importante en la semántica porque determina cómo y cuándo se resuelven los nombres en el código.

La ligadura estática se establece en tiempo de compilación y no cambia en tiempo de ejecución. Es más eficiente porque el compilador ya conoce las referencias. Un ejemplo en C++ es `int x = 10;`, donde `x` se asocia con una dirección de memoria antes de la ejecución. También ocurre en polimorfismo cuando no se usa `virtual`, como en:

```
cpp
class A { public: void show() { cout << "Clase A"; } };
class B : public A { public: void show() { cout << "Clase B"; } };
int main() { A obj; obj.show(); } // Se resuelve en compilación: "Clase A"
```

En lenguajes como Python y JavaScript, las variables pueden cambiar de tipo en tiempo de ejecución, lo que es un ejemplo de ligadura dinámica. En Python:

```
python
x = 10 # x es un entero
x = "hola" # x ahora es un string, la ligadura se resuelve en ejecución
```


Práctica 4 - Variables

1)

Identificador	Tipo	L-Valor	R-Valor	Alcance	T.vida
A	Entero	Automática	Indefinido	4-16	1-16
p	Puntero	Automática (variable normal, de tipo puntero)	Nil	5-16	1-16
p^	Puntero o variable apuntada	Dinámica	Indefinido	5-16	7-15

2)a)

- **Inicialización por defecto:** Enteros se inicializan en 0, caracteres en blanco, etc.
- **Inicialización en la declaración:** `int i = 0; int j = 2;`
- Si no es inicializada, **Ignorar el problema:** Toma como valor inicial lo que hay en memoria (la cadena de bits asociados al área de almacenamiento. Puede llevar a errores.

2)b)

Lenguaje	Inicialización por defecto	Inicialización en la declaración	No inicialización
Java	Sí, en variables de instancia. No en variables locales	Sí	No permitida en variables locales. El compilador obliga a inicializar
C	No por defecto en variables locales. Sí para globales/estáticas: en 0	Sí	Sí, puede usarse sin inicialización. Valor basura
Python	No hay inicialización por	Sí	No permitido. Tira error

		defecto		
	Ruby	No hay inicialización por defecto	Sí	No permitido. Tira error

3)

- **Variable estática:** Se hace en compilación cuando se carga el programa en memoria en zona de datos y perdura hasta fin de la ejecución.
 - Existe durante toda la ejecución del programa.
 - `static int x = 0; ©`
- **Variable automática:** Se crean al entrar a un bloque (como función) y se destruyen al salir. También se las llama locales. Su l-valor puede definirse en tiempo de compilación si el tamaño es fijo.
 - Existe sólo durante la ejecución del bloque o función.
- **Variable dinámica:** Una variable dinámica es una variable que se crea y se destruye en tiempo de ejecución. En la mayoría de los lenguajes de programación, son los punteros.
- **Variable semidinámica:** Una variable semidinámica es aquella cuyo tamaño puede cambiar durante la ejecución del programa, pero solo se puede ajustar su tamaño en ciertos puntos de control predefinidos. Un ejemplo de una variable semidinámica en ADA podría ser una matriz cuyo tamaño se puede ajustar en tiempo de ejecución, pero solo en un punto de control específico del programa.

En el lenguaje de programación **C**, se pueden clasificar las variables según su l-valor de la siguiente manera:

1. **Variable estática:** son variables que se almacenan en la memoria estática y mantienen su valor entre llamadas a funciones. Estas variables se declaran con la palabra clave "static" y se inicializan automáticamente en cero.

2. **Variable automática:** son variables que se almacenan en la memoria de la pila y se eliminan automáticamente al salir de su ámbito de definición. Estas variables se declaran sin la palabra clave "static" y su vida útil está limitada a su función o bloque de código.

3. **Variable dinámica:** son variables que se asignan en tiempo de ejecución y se almacenan en la memoria dinámica. Estas variables se crean utilizando las funciones "malloc" o "calloc" y se liberan utilizando la función "free". Estas variables tienen una vida útil que se extiende más allá de la función o bloque de código en el que se crean.

En el lenguaje de programación **Ada**, las variables se clasifican según su valor de la siguiente manera:

1. **Variable estática:** se definen usando la palabra clave "constant" o "static" y se mantiene en memoria durante toda la ejecución del programa.

2. **Variable automática o semiestática:** se definen dentro de un bloque y se eliminan de la memoria cuando el bloque termina.

3. **Variable dinámica:** se asignan en tiempo de ejecución usando la palabra clave "new". Estas variables se mantienen en la memoria hasta que se libera explícitamente usando la palabra clave "delete".

4. **Variable semidinámica:** una matriz cuyo tamaño se puede ajustar en tiempo de ejecución, pero sólo en un punto de control específico del programa.

4)a) Una **variable local** es una variable declarada dentro de una función o bloque de código y su ámbito de alcance se limita a la misma. Variables creadas dentro de una unidad.

Una **variable global** es una variable declarada fuera de cualquier función o bloque de código y su ámbito de alcance se extiende a todo el programa. Variables creadas en el programa principal.

4)b) Significa que la **variable es declarada dentro de una función (es local)**, pero **su espacio en memoria no se crea y destruye cada vez que se llama la función**, sino que **se mantiene (es persistente)** durante toda la ejecución del programa.

Sí es posible, una **variable local puede ser estática respecto a su L-valor**. En algunos lenguajes como C y C++, se pueden definir variables locales como estáticas para mantener su valor entre llamadas sucesivas a una función.

4)c) No necesariamente. Una variable global no siempre es estática, aunque depende del lenguaje utilizado.

En algunos lenguajes como C sí se cumple esta propiedad. Mientras que en otros lenguajes como Python no. Su tiempo de vida está ligado al módulo donde se definen, pero no se almacenan en memoria estática.

4)d) La principal diferencia entre una variable estática respecto a su L-valor y una constante es que el valor de una constante no puede ser modificado una vez que se ha definido, mientras que el valor de una variable estática sí puede ser modificado durante la ejecución del programa.

El valor de una constante se define una sola vez y no se puede modificar en tiempo de ejecución. Una variable estática se define en el código y su valor se mantiene en memoria durante toda la ejecución del programa, pero su valor se puede modificar en cualquier momento.

5)a) La clasificación de las constantes en Ada en numéricas y comunes se debe a que las constantes numéricas son aquellas que se refieren a valores numéricos, mientras que las constantes comunes son aquellas que pueden contener caracteres y símbolos, como cadenas de texto.

5)b) El momento de ligadura de las constantes en Ada es durante la compilación del programa, es decir, en tiempo de compilación.

- el momento de ligadura de la constante "H" es cuando se le asigna el valor 3.5
- el momento de ligadura de la constante "I" es cuando se le asigna el valor 2

- el momento de ligadura de la constante "K" es cuando se calcula su valor mediante la multiplicación de "H" e "I".

Por lo tanto, el momento de ligadura de "K" es después del momento de ligadura de "H" e "I" durante el proceso de compilación. Como todas las constantes se definen en tiempo de compilación, su valor se conoce antes de que se ejecute el programa.

6) Sí, en principio tendría el mismo comportamiento ya que en ambos casos se estaría almacenando la variable en memoria durante toda la ejecución del programa. De la única forma que podría variar es si en ningún momento del programa se llama a func1(). De esta forma, la segunda versión podría tener un comportamiento distinto en memoria en algunos casos específicos.

7)

Identificadores globales: Variables estáticas de clase, que se declaran a nivel de clase y pueden ser accedidos sin necesidad de crear una instancia de la clase:

- `public static int cantTotalPersonas` -> clase Persona
- `public static int nro` -> clase Domicilio

Variables de instancia: No estáticas, son variables de instancia que se declaran dentro de la clase y solo pueden ser accedidas a través de una instancia de la clase.

- `id` en las clases Persona y Domicilio
- `nombreApellido` en la clase Persona
- `domicilio` en la clase Persona
- `dni` en la clase Persona
- `fechaNac` en la clase Persona
- `calle` en la clase Domicilio
- `loc` en la clase Domicilio

Variables locales: Variables que se declaran dentro de un método y solo pueden ser accedidos dentro del mismo.

- `edad` en el método `getEdad()` de la clase Persona
- `fN` en el método `getEdad()` de la clase Persona

La diferencia entre variables globales, variables de instancia y variables locales en Java se basa en su alcance y su duración.

Las **variables globales** son visibles en toda la clase y tienen una duración de toda la vida del programa, mientras que las **variables de instancia** son específicas de cada objeto y duran tanto como el objeto exista. Por último, las **variables locales** solo son visibles en el método o bloque de código en el que se declaran y tienen una duración limitada a la ejecución de ese método o bloque.

8)

a)

- Tiempo de vida de variable **i**: 1-15
- Tiempo de vida de variable **h**: 1-15
- Tiempo de vida de variable **mipuntero**: 1-15
- Tiempo de vida de variable **^mipuntero**: 9-12

b)

- Alcance de variable **i**: 5-15
- Alcance de vida de variable **h**: 6-15
- Alcance de vida de variable **mipuntero**: 4-15
- Alcance de vida de variable **^mipuntero**: 4-15

c) No, no representa un error ya que es correcto el contenido que tiene. Como la variable "mipuntero" apunta a la variable i que también es un entero, es lo mismo que hacer $h := i+i$. Por otro lado, no hay problema con el **dispose** ya que se realiza luego de almacenar el valor en h.

d) Sí, representa un error ya que se intenta hacer la resta luego de hacer el **dispose** de "mipuntero". Se está tratando de hacer una resta entre un entero y un null, lo que genera un error.

e) Sí, la entidad faltante es el programa principal. En el caso de Pascal, el programa principal tiene alcance global y su tiempo de vida es desde su declaración en la línea 6 hasta el final del programa en la línea 15.

f)

- Tipo de L-valor variable **i**: Automática
- Tipo de L-valor variable **h**: Automática
- Tipo de L-valor variable **mipuntero**: Automática
- Tipo de L-valor variable **^mipuntero**: Dinámica

9) a)

```
c

int* guardarValor() {
    static int x = 42; // Vive durante todo el programa
    return &x;         // Pero solo es accesible dentro de esta función
}

int main() {
    int* puntero = guardarValor();
    printf("%d\n", *puntero); // Imprime 42
    return 0;
}
```

- La **variable x** está **dentro de la función guardarValor**, o sea su **alcance (scope)** es local.
- Pero al estar declarada como **static**, su **tiempo de vida dura todo el programa**.
- Por lo tanto, aunque ya **salí del bloque donde fue declarada**, la variable **sigue existiendo en memoria**.

9) b) Variable dinámica con dispose. El alcance es el mismo que tiene la variable de tipo puntero (no la apuntada), mientras que el tiempo de vida es desde el new() hasta el dispose().

9) c) Variable local automática de una función. Tiempo de vida y alcance hasta que termine la función.

10) Si, se puede asegurar que el tiempo de vida y el alcance de la variable "c" es siempre todo el procedimiento en el que se encuentra definida ya que no existen subprocesos internos que puedan modificar el alcance o el tiempo de vida de la misma.

11)

- I) Falso. No se usa para referenciar la variable.
- II) Falso. El tipo de dato no tiene relación con el alcance.
- III) Falso. No está relacionado con el lugar de memoria.
- IV) Verdadero.

Definición correcta de tipo de dato: Un tipo de dato es una abstracción que define el conjunto de valores que una variable puede tomar y el conjunto de operaciones que se pueden realizar sobre esos valores.

12)

Identificador	Tipo	r-valor	Alcance	Tiempo de vida
Main(2)	-	-	3-29	1-29
a (4)	Automática	Basura	5-29	1-29
n(4)	Automática	Basura	5-29	1-29
p(4)	Automática	Basura	5-11 y 23-29	1-29
v1(5)	Automática	Basura	6-29	1-29
c1(6)	Automática	Basura	7-10 y 23-29	1-29
Uno(7)	-	-	8-29	7-22
v2(9)	Semidinámica	Basura	10-22	7-22

c1(10)	Automática	Basura	11-22	7-22
c2(10)	Automatcia	Basura	11-22	7-22
p(11)	Automática	Nil	12-22	7-22
p^	Dinámica	Basura	12-22	15-18
q(11)	Automática	Nil	12-22	7-22
q^	Dinámica	Basura	12-22	16-20

13)a) No, el nombre de una variable no puede condicionar el tiempo de vida de la misma, el concepto de nombre es lo que se utiliza para referenciar a la variable mientras que el tiempo de vida de la misma es el periodo de tiempo que la variable está alocada en memoria y su binding existe.

13)b) Sí, el nombre de una variable puede condicionar su alcance, dependiendo de si hay otra variable igual en alguna parte del programa que la "pise". Por ejemplo, una variable global P y en una función se declara una variable local P. En este caso, si la variable global P tendría otro nombre, podría ser utilizada dentro de la función.

13)c) No, el nombre no condiciona el r-valor de la misma, este último concepto es definido como el valor codificado almacenado en la locación asociada a la variable, es decir, a su l-valor. Nosotros podemos acceder al r-valor gracias al nombre ya que este es el que nos sirve para poder referenciar a la misma.

13)d) No, el nombre de una variable no condiciona el tipo de la misma, el tipo normalmente se define en la declaración de la variable y es el que condiciona qué conjunto de valores y operaciones podemos hacer con la variable pero esta puede tener cualquier nombre y esto no va a interferir con el nombre de la misma.

14)

Identificador	Tipo	r-valor	Alcance	Tiempo de vida
v1(1)	Automática	0	2-4 y 9-12 y 21-23	1-28

*a(2)	Automática	Null	3-16	1-28
a^	Dinámica	Basura	3-16	15-16
fun2(3)	-	-	4-16	3-8
v1(4)	Automática	Basura	5-8	3-8
y(4)	Automática	Basura	5-8	3-8
main(9)	-	-	10-16	9-16
var3(10)	Estática	0	11-16	<1-28>
v1(12)	Automática	Basura	13-16	9-16
y(12)	Automática	Basura	13-16	9-16
var1(14)	Automática	Basura	15-16	13-15
aux(17)	Estática	0	18-25	<1-28>
v2(18)	Automática	0	7 y 12-16 y 19-28	1-28
fun2(19)	-	-	20-28	<1-28>
fun3(24)	-	-	25-28	24-28
aux(25)	Automática	Basura	26-28	24-28

15)

Modificador	Javascript	Python
"const"	Declara una variable con un valor constante, que no puede ser reasignado. Debe ser inicializado al declararlo.	No hay un equivalente directo en Python. Las variables en Python pueden ser reasignadas, pero una convención similar a "const" es usar nombres de variables en mayúsculas para indicar que no deben ser modificadas.
"var"	Declara una variable de alcance global o de función. No tiene restricciones de	No hay un equivalente directo en Python. En Python, todas las variables

	reasignación o redeclaración. Es menos comúnmente utilizado desde la introducción de let y const.	declaradas dentro de una función tienen un alcance local. Variables definidas fuera de las funciones tienen un alcance global.
"let"	Declara una variable de bloque con un alcance limitado al bloque de código en el que está definida. Puede ser reasignada pero no redeclarada en el mismo ámbito.	No hay un equivalente directo en Python. Sin embargo, las variables definidas en un bloque de código (como dentro de un bucle for o if) tienen un alcance limitado a ese bloque.
Ausencia	Si una variable se declara sin ninguno de los modificadores anteriores, su comportamiento depende del contexto: en el caso de variables declaradas fuera de cualquier función, se comportan como variables globales (var); si se declaran dentro de una función, se comportan como variables de función (var). Desde ECMAScript 6 (ES6), se considera una mala práctica declarar variables sin especificar un modificador.	Las variables sin declarar son consideradas variables locales en Python, y arrojarán un error si se intentan utilizar antes de ser inicializadas. Es una buena práctica inicializar todas las variables antes de usarlas.

Práctica 5 - Pilas de Ejecución

1)

Head (prog principal)
Pto retorno
EE (enlace estático)
ED (enlace dinámico)
Variables...
...
Parámetros ...
....
Procedimientos
....
Funciones ...
....
Valor de retorno

Registro de activación: Estructura de datos fundamental en la ejecución de programas, especialmente cuando se utilizan procedimientos o funciones. Su utilidad principal es mantener la información necesaria para la ejecución de un procedimiento o función, permitiendo la gestión eficiente de la pila y la correcta manipulación de variables locales, parámetros y contexto.

- **Head:** Marca el inicio del registro. Puede incluir metadatos como el nombre de la subrutina o información del programa principal. Generalmente contiene:
 - **Current:** Dirección base del registro de activación de la unidad que se este ejecutando actualmente.
 - **Free:** Próxima dirección libre en la pila.
- **Punto de retorno:** Cuando una rutina llama a otra y esta última termina, el punto de retorno es la dirección de memoria donde continúa la ejecución.
- **Enlace estático:** Puntero a la dirección base del registro de activación de la rutina que estáticamente la contiene.
- **Enlace dinámico:** Puntero a la dirección base del registro de activación de la rutina llamadora.
- **Variables:** Se enumeran las variables que conforman la unidad y se van reemplazando los valores de acuerdo a la ejecución del programa
- **Procedimientos y funciones:** Se enumeran los identificadores de los proc y funciones que contiene la unidad.
- **Valor de retorno:** Los valores retornados por las funciones que desde esta unidad se llaman a ejecutar deberán ser escritos en esta dirección de memoria.

2)

Código	Cadena estática	Cadena dinámica
--------	-----------------	-----------------

Program Main

Var a: array[1..10] of integer;

x,y,z:integer

Procedure A ()

var y,t: integer;

begin

a(1):= a(1)+1;z:=z+1;

t:=1; y:=2;

B(); a(y):=a(y)+3; y:=y+1;

If z=11 Then Begin

a(z-1):=a(z-2) + 3;

z:=z-4;

a(z-y):=a(z) - a(y) + 5;

End;

end;

Function t():integer

begin

y:=y+1; z:=z-6;

return(y+x);

end;

Procedure B()

var d:integer;

Procedure I ()

begin

x:=0; x:=x+6;

end;

begin

x:=x+t; d:=0;

*** Reg Activ Main

*1 Pto retorno

A(1)= 1 -> 2

A(2)= 2 -> 5

A(3)= 3

A(4)= 4

A(5)= 5

A(6)= 6

A(7)= 7

A(8)= 8

A(9)= 9

A(10)= 10

X= 1..10 -> 5 -> 12 -> 0 -> 6 -> 5 -> 0

-> 6 -> 5 -> 0 -> 6 -> 5 -> 1..10

Y= 1 -> 2

Z= 10 -> 11 -> 5

Procedure A

Function t

Procedure B

VR

*2 ***Reg Activ A

Pto Retorno

EE (*1)

ED (*1)

Y = 2 -> 3

T = 1

VR

*3*** Reg Activ B

Pto Retorno

EE = (*1)

ED (*2)

d = 0 -> 2 -> 4 -> 6

Procedure I

VR .. ¿? ..

*4*** Reg Activ t

Pto Retorno

EE (*1)

ED (*3)

VR .. 7 ..

*** Reg Activ Main

*1 Pto retorno

A(1)= 1 -> 2 -> 5

A(2)= 2 -> 5

A(3)= 3

A(4)= 4 -> 9

A(5)= 5

A(6)= 6

A(7)= 7

A(8)= 8

A(9)= 9

A(10)= 10 -> 12

x= 1..10 -> 5 -> 6 -> 0 -> 6 -> 5 -> 0

-> 6 -> 5 -> 0 -> 6 -> 5 -> 1..10

y= 1 -> 2

z=10 -> 11 -> 7

Procedure A

Function t

Procedure B

VR

*2 ***Reg Activ A

Pto Retorno

EE (*1)

ED (*1)

y = 2 -> 3

t = 1

VR

*3 *** Reg Activ B

Pto Retorno

EE (*1)

ED (*2)

d = 0 -> 2 -> 4 -> 6

Procedure I

VR .. ¿? ..

*4 *** Reg Activ I

Pto Retorno

EE (*3)

ED (*3)

VR .. ¿? ..

<pre> while x>d do begin l(); x:=x-1; d:=d + 2; end; end; begin For x:=1 To 10 do a(x):=x; x:=5; y:=1; z:=10; A(); For x:=1 To 10 do write(a(x),x); end. </pre>	<pre> *5*** Reg Activ I Pto Retorno EE (*3) ED (*3) VR .. ¿? .. *6*** Reg Activ I Pto Retorno EE (*3) ED (*3) VR .. ¿? .. *7*** Reg Activ I Pto Retorno EE (*3) ED (*3) VR .. ¿? .. Imprime: 2,1 5,2 3,3 4,4 5,5 6,6 7,7 8,8 9,9 10,10 </pre>	<pre> *5 *** Reg Activ I Pto Retorno EE (*3) ED (*3) VR .. ¿?.. *6 *** Reg Activ I Pto Retorno EE (*3) ED (*3) VR .. ¿?.. Imprime: 5,1 5,2 3,3 9,4 5,5 6,6 7,7 8,8 9,9 12,10 </pre>
--	--	---

3)

Código	Cadena estática	Cadena dinámica
--------	-----------------	-----------------

<pre> 1 PROGRAM P1; 2 var 3 a:integer; 4 b:char; 5 c: array[1..10] of integer 6 7 Procedure PP1; 8 var 9 a:char; 10 p:integer; 11 Function x: integer; 12 var 13 z:integer; 14 begin 15 a:="j"; 16 z=-1; 17 return z; 18 end; 19 Begin 20 p:=x; 21 write(a); 22 p:=x+3; 23 c[p]=8; 24 p:=x+2; 25 c[p]=x; 26 end; 27 28 Procedure x; 29 var 30 b:char; 31 Procedure PP2; 32 Begin 33 write("para qué estoy aquí?"); 34 end; 35 Begin 36 a:=1; 37 c[a]:=4; 38 b:="a"; 39 write(concat(c[1],b)); /*concat convierte a string los 40 parámetros, concatena y retorna un string;*/ 41 PP1(); 42 b:="b"; 43 write(concat(c[5],b)); /*concat convierte a string los 44 parámetros, concatena y retorna un string;*/ 45 End; 46 47 BEGIN 48 a:=3; 49 b:="c"; 50 for a:=3 to 10 do 51 begin 52 c[a]:=2*a; 53 end; 54 x; 55 write(b); 56 write(a); 57 for a:=1 to 10 do 58 write(c[a]-3); 59 END. </pre>	<pre> *** Reg Activ Main *1 Pto retorno a = 3, 4, 1..10 b = "c" c(1) = 4, -1 c(2) = 8 c(3) = 6 c(4) = 8 c(5) = 10 c(6) = 12 c(7) = 14 c(8) = 16 c(9) = 18 c(10) = 20 Procedure PP1 Procedure x VR *2 ***Reg Activ x Pto Retorno EE (*1) ED (*1) b = "a", "4", "b", "10" Procedure PP2 VR *3*** Reg Activ PP1 Pto Retorno EE (*1) ED (*2) a = "j" p = -4, 2, 1 Function x VR .. ¿? .. *4*** Reg Activ x Pto Retorno EE (*4) ED (*4) z = -1 VR .. -1 .. </pre>	<pre> *** Reg Activ Main *1 Pto retorno a = 3, 4, 1..10 b = "c" c(1) = 4, -1 c(2) = 8 c(3) = 6 c(4) = 8 c(5) = 10 c(6) = 12 c(7) = 14 c(8) = 16 c(9) = 18 c(10) = 20 Procedure PP1 Procedure x VR *2 ***Reg Activ x Pto Retorno EE (*1) ED (*1) b = "a", "4", "b", "10" Procedure PP2 VR *3*** Reg Activ PP1 Pto Retorno EE (*1) ED (*2) a = "j" p = -4, 2, 1 Function x VR .. ¿? .. *4*** Reg Activ x Pto Retorno EE (*4) ED (*4) z = -1 VR .. -1 .. </pre>
---	---	---

	<div><div><div>*5*** Reg Activ x</div><div>Pto Retorno</div><div>EE (*4)</div><div>ED (*4)</div><div>z = -1</div><div>VR..¿?..</div></div><div><div>*6*** Reg Activ x</div><div>Pto Retorno</div><div>EE (*4)</div><div>ED (*4)</div><div>z = -1</div><div>VR..¿?..</div></div></div>	<div><div><div>*5*** Reg Activ x</div><div>Pto Retorno</div><div>EE (*4)</div><div>ED (*4)</div><div>z = -1</div><div>VR..¿?..</div></div><div><div>*6*** Reg Activ x</div><div>Pto Retorno</div><div>EE (*4)</div><div>ED (*4)</div><div>z = -1</div><div>VR..¿?..</div></div></div>	
--	---	---	--

Código	Cadena estática	Cadena dinámica
--------	-----------------	-----------------

```

PROGRAM P1;
var
  a:integer;
  b:char;
  c: array[1..10] of integer

Procedure PP1;
  var
    a:char;
    p:integer;
  Function x: integer;
    var
      z:integer;
  begin
    a:="j";
    z=-1;
    return z;
  end;
Begin
  p:=x;
  write(a);
  p:=x+3;
  c[p]=8;
  p:=x+2;
  c[p]=x;
end;

Procedure x;
  var
    b:char;
  Procedure PP2;
  Begin
    write("para qué estoy aquí?");
  end;
Begin
  a:=1;
  c[a]:=4;
  b:="a";

```

```

*** Reg Activ Main
*1 Pto retorno
a = 3 -> 3..10 -> 1 -> 1..10
b = "c"
c(1) = 4 -> -1
c(2) = 8
c(3) = 6
c(4) = 8
c(5) = 10
c(6) = 12
c(7) = 14
c(8) = 16
c(9) = 18
c(10) = 20
Procedure PP1
Procedure x
Valor de Retorno

```

```

*2 *** Reg Activ x
Pto Retorno
EE (*1)
ED (*1)
b = "a" -> "b"
Procedure PP2
VR .. ¿? ..

```

```

*3 *** Reg Activ PP1
Pto Retorno
EE (*1)
ED (*2)
a = "j" -> "j" -> "j" -> "j"
p = -1 -> 2 -> 1
Function x
VR: -1, -1, -1, -1

```

```

*4 *** Reg Activ x
Pto Retorno
EE (*3)
ED (*3)
z = -1
VR

```

```

*** Reg Activ Main
*1 Pto retorno
a = 3 -> 3..10 -> 1 -> 1..10
b = "c"
c(1) = 4 -> -1
c(2) = 8
c(3) = 6
c(4) = 8
c(5) = 10
c(6) = 12
c(7) = 14
c(8) = 16
c(9) = 18
c(10) = 20
Procedure PP1
Procedure x
Valor de retorno

```

```

*2 *** Reg Activ x
Pto Retorno
EE (*1)
ED (*1)
b = "a" -> "b"
Procedure PP2
VR .. ¿? ..

```

```

*3 *** Reg Activ PP1
Pto Retorno
EE (*1)
ED (*2)
a = "j" -> "j" -> "j" -> "j"
p = -1 -> 2 -> 1
Function x
VR: -1, -1, -1, -1

```

```

*4 *** Reg Activ x
Pto Retorno
EE (*3)
ED (*3)
z = -1
VR

```

<pre> write(concat(c[1],b)); /*concat convierte a string los parámetros, concatena y retorna un string;*/ PP1(); b:="b"; write(concat(c[5],b)); /*concat convierte a string los parámetros, concatena y retorna un string;*/ End; BEGIN a:=3; b:="c"; for a:=3 to 10 do begin c[a]:=2*a; end; x; write(b); write(a); for a:=1 to 10 do write(c[a]-3); END. </pre>	<pre> *5 *** Reg Activ x Pto Retorno EE (*3) ED (*3) z = -1 VR *6 *** Reg Activ x Pto Retorno EE (*3) ED (*3) z = -1 VR *7 *** Reg Activ x Pto Retorno EE (*3) ED (*3) z = -1 VR Imprime: 4a j 10b c 1 -4 5 3 5 7 9 11 13 15 17 </pre>	<pre> *5 *** Reg Activ x Pto Retorno EE (*3) ED (*3) z = -1 VR *6 *** Reg Activ x Pto Retorno EE (*3) ED (*3) z = -1 VR *7 *** Reg Activ x Pto Retorno EE (*3) ED (*3) z = -1 VR Imprime: 4a j 10b c 1 -4 5 3 5 7 9 11 13 15 17 </pre>
---	---	---

	Código	Cadena estática	Cadena dinámica	
--	--------	-----------------	-----------------	--

<pre> 1 Program Main; 2 Var x, y, z:integer; 3 a, b: array[1..6] of integer; 4 5 Procedure B; 6 var 7 y,x: integer; 8 Procedure C; 9 var c:integer; 10 begin 11 y:= y + 2; c:=2; 12 a(x):=a(x)*y; 13 if (y >7) then 14 b(y-6)=b(4)*2+b(y-6); 15 D; 16 end; 17 begin 18 x:=2; y:= x + 3; 19 C; x:= x + 1; write (x,y); 20 End; 21 22 Procedure D; 23 begin 24 x:= c + 5 + x; 25 y:= y + 2; 26 end; 27 28 Function C: integer; 29 begin 30 b(x):= b(x) + 1; 31 x:= x + 1; 32 a(y):=a(y)+b(x)+3; 33 a(x+2)=a(x) + 2; 34 return b(x); 35 end 36 37 begin 38 x:= 1; Y:= 2; 39 for z:=1 to 6 do begin 40 a(z):= z; 41 b(z):= z + 2; 42 end; 43 B; 44 for z:= to 6 do write (a(z), b(z)); 45 end.</pre>	<p>*** Reg Activ Main</p> <p>*1 Pto retorno</p> <p>x = 1 -> 2 -> 11 y = 2 -> 4 z = 1..6 -> 1..6 a(1) = 1 a(2) = 2 -> 14 -> 21 a(3) = 3 a(4) = 4 -> 23 a(5) = 5 a(6) = 6 b(1) = 3 -> 4 b(2) = 4 b(3) = 5 b(4) = 6 b(5) = 7 b(6) = 8 Procedure B Procedure D Function C Valor de Retorno</p> <p>*2*** Reg Activ B</p> <p>Pto Retorno</p> <p>EE (*1) ED (*1) y = 5 -> 7 x = 2 -> 3 Procedure C VR</p> <p>*3 *** Reg Activ C</p> <p>Pto Retorno</p> <p>EE (*2) ED (*2) c = 2 VR</p> <p>*4 *** Reg Activ D</p> <p>Pto Retorno</p> <p>EE (*1) ED (*3)</p>	<p>*** Reg Activ Main</p> <p>*1 Pto retorno</p> <p>x = 1 y = 2 z = 1..6 -> 1...6 a(1) = 1 a(2) = 2 -> 14 a(3) = 3 a(4) = 4 a(5) = 5 a(6) = 6 b(1) = 3 b(2) = 4 b(3) = 5 b(4) = 6 b(5) = 7 b(6) = 8 Procedure B Procedure D Function C Valor de Retorno</p> <p>*2*** Reg Activ B</p> <p>Pto Retorno</p> <p>EE (*1) ED (*1) y = 5 -> 7 -> 9 x = 2 -> 9 -> 10 Procedure C VR</p> <p>*3*** Reg Activ C</p> <p>Pto Retorno</p> <p>EE (*2) ED (*2) c = 2 VR</p> <p>*4*** Reg Activ D</p> <p>Pto Retorno</p> <p>EE (*1) ED (*3)</p>
--	---	--

	VR 4 *5 *** Reg Activ C Pto Retorno EE (*1) ED (*4) VR Imprime: 3,7 1,4 21,4 3,5 23,6 5,7 6,8	VR Imprime: 10,9 1,3 14,4 3,5 4,6 5,7 6,8	
--	--	--	--

No sería lo mismo poner $x := x + c + 5$, ya que en la estática cuando se pone “c” se llama a la función y la misma modifica el valor de la variable x. Con esta sentencia estaría poniendo el valor en x después de habla sumado.