



UNIVERSIDAD  
DE GRANADA

# Tema 3: Compilación y enlazado de programas

Fundamentos del Software

1º Grado en Informática

Rosana Montes - [rosana@ugr.es](mailto:rosana@ugr.es)

# Objetivos

- ♦ Justificar la existencia de los lenguajes de programación.
- ♦ Conocer el proceso de traducción.
- ♦ Diferenciar entre compilación e interpretación.
- ♦ Identificar los elementos que intervienen en la gestión de memoria.
- ♦ Conocer las necesidades de memoria de los procesos.
- ♦ Conocer el proceso de enlazado de programas.
- ♦ Conocer las diferencias entre enlace estático y dinámico.
- ♦ Reconocer diferentes tipos de bibliotecas.

## 3.1. Lenguajes de Programación

### Concepto de Lenguaje de Programación [Prie06] (pp. 581-591)

Conjunto de símbolos y de reglas para combinarlos, que se usan para expresar algoritmos.

Características:

- Independiente de la arquitectura física del computador.
- Algo expresado en un lenguaje de alto nivel utiliza notaciones más cercanas a las habituales en el ámbito en que se usan.
- Una sentencia en un lenguaje de alto nivel da lugar, tras el proceso de traducción, a varias instrucciones en lenguaje máquina.

### 3.1. Lenguajes de Programación

Una sentencia en un lenguaje de alto nivel da lugar, tras el proceso de traducción, a varias instrucciones en lenguaje máquina.

Lenguaje de Alto Nivel	Lenguaje Ensamblador	Lenguaje Máquina
A=B+C	LDA 0, 4, 3	021404
	LDA 2, 3, 3	031403
	ADD 2, 0	143000
	STA 0, 5, 3	041405

Ensamblador	C
mov Ax, b mov Bx, c comp Ax, Bx eq Cx, 0 jump 07 sum Ax, Bx jump 08 mult Ax, Bx exit	if ( b==c ) b = b+c; else c = b*c;

## 3.1. Lenguajes de Programación

### Concepto de Lenguaje de Programación [Prie06] (pp. 581-591)

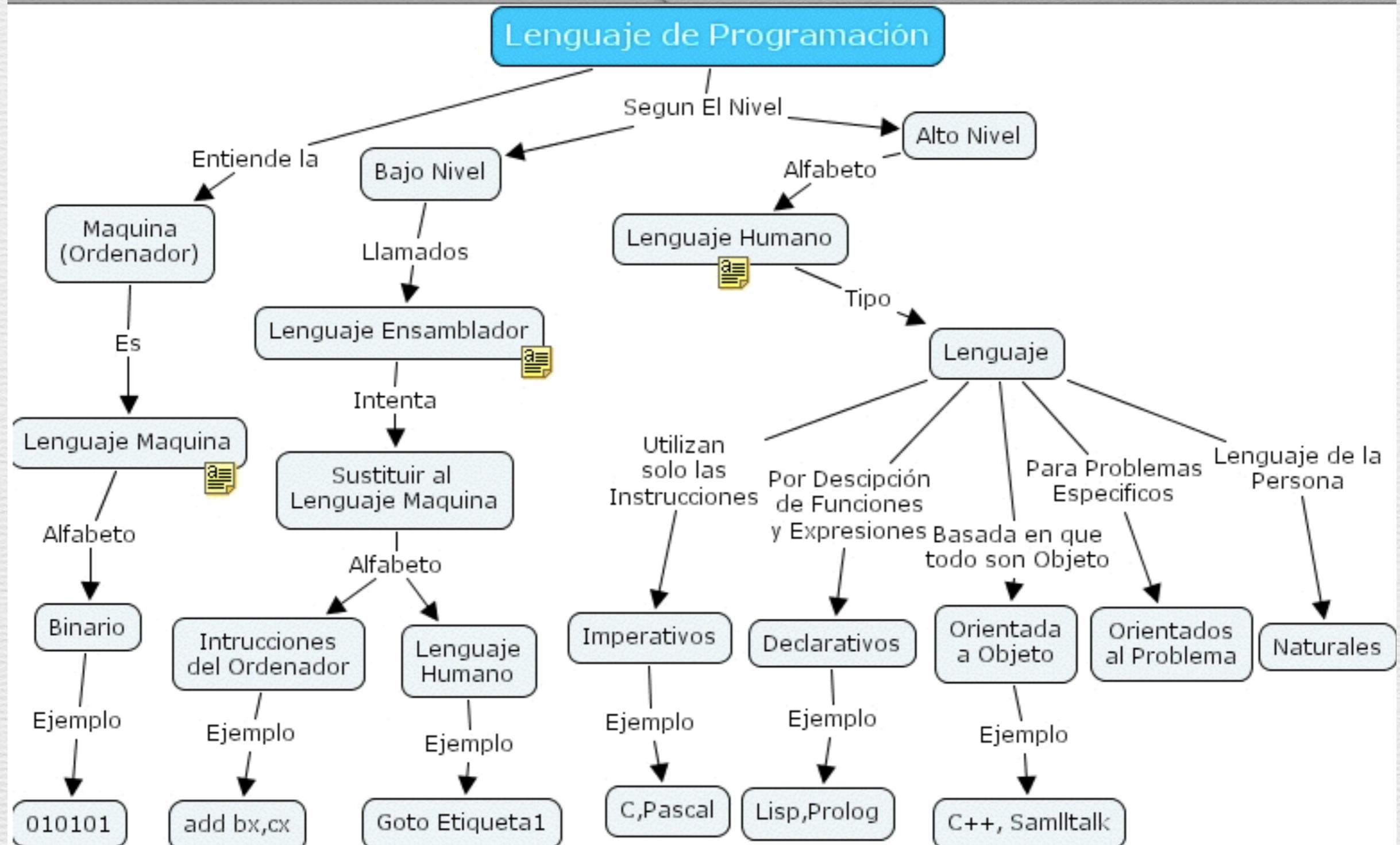
Otras características:

- Es necesario un proceso de traducción de fuente a código interpretable por el procesador.
- Estándares de portabilidad (ej. ANSI): permiten que el lenguaje máquina resultante sea ejecutable en otra arquitectura (plataformas).
- Permite la definición de variables, funciones, procedimientos, clases, etc.
- Ofrece funciones aritmético-lógicas, de manejo de texto, de manejo de ficheros, etc.
- Existen convenios de notación, comentarios, tabulación... para mejorar la legibilidad del código fuente.

# Lenguajes

Lenguaje	Principal área de aplicación	Compilado/interpretado
ADA	Tiempo real	Lenguaje compilado
BASIC	Programación para fines educativos	Lenguaje interpretado
C	Programación de sistema	Lenguaje compilado
C++	Programación de sistema orientado a objeto	Lenguaje compilado
Cobol	Administración	Lenguaje compilado
Fortran	Cálculo	Lenguaje compilado
Java	Programación orientada a Internet	Lenguaje intermediario
MATLAB	Cálculos matemáticos	Lenguaje interpretado
APL	Cálculos matemáticos	Lenguaje interpretado
LISP	Inteligencia artificial	Lenguaje intermediario
Pascal	Educación	Lenguaje compilado
PHP	Desarrollo de sitios web dinámicos	Lenguaje interpretado
Prolog	Inteligencia artificial	Lenguaje interpretado
Perl	Procesamiento de cadenas de caracteres	Lenguaje interpretado
R	Cálculo científico / estadístico	Lenguaje interpretado

# 3.1. Lenguajes de Programación



[http://grupoorion.unex.es:8001/rid=1111146176050\\_2119398457\\_9214/Lenguaje%20Programacion%20\(ManuelGomez\).cmap](http://grupoorion.unex.es:8001/rid=1111146176050_2119398457_9214/Lenguaje%20Programacion%20(ManuelGomez).cmap)

This Concept Map was created with  
IHMC CmapTools

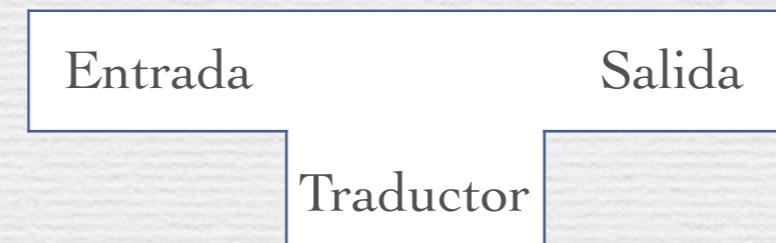


## 3.2. Construcción de Traductores

**Traductor** es un programa que recibe como entrada un texto en un lenguaje de programación concreto y produce, como salida, un texto en lenguaje [máquina] equivalente.

**Entrada:** lenguaje fuente, que define a una *máquina virtual*.

**Salida:** lenguaje objeto, que define a una máquina real.

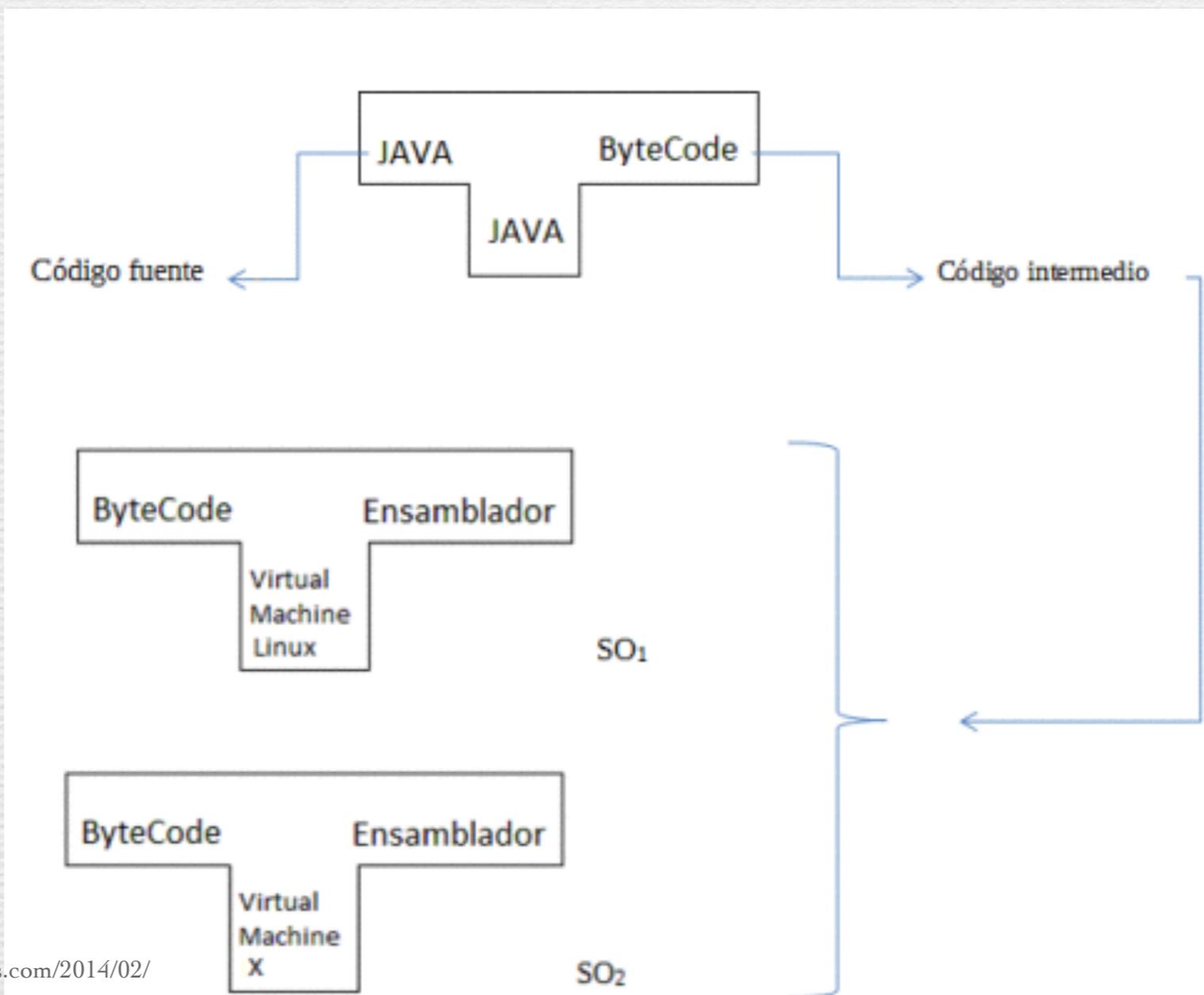


Dos aproximaciones a la traducción:

- Compilación (por el compilador).
- Interpretación (por el intérprete).

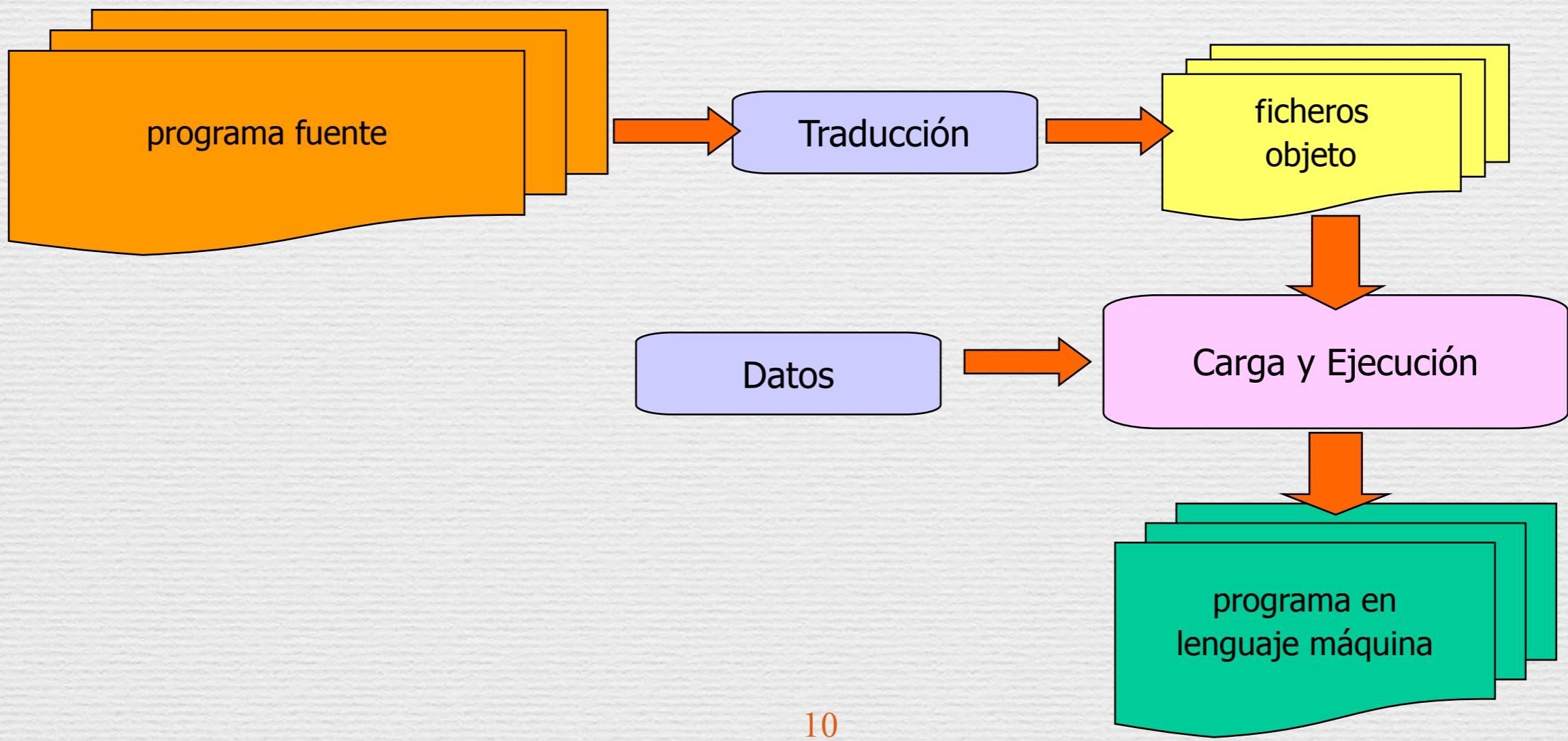
# Traductor

**Java** es un lenguaje que combina compilación e interpretación y por eso es *multiplataforma*.



# Compilador

Traduce la especificación de entrada a lenguaje máquina incompleto y con instrucciones máquina incompletas. Requiere de un complemento denominado **ensamblador**.



# Compilador

El **enlazador** (*linker*) realiza el enlazado de los programas completando las instrucciones máquina necesarias.

- ♦ Añade rutinas binarias de funcionalidades no programadas directamente en el programa fuente (bibliotecas).
- ♦ Genera un programa ejecutable para la máquina real.

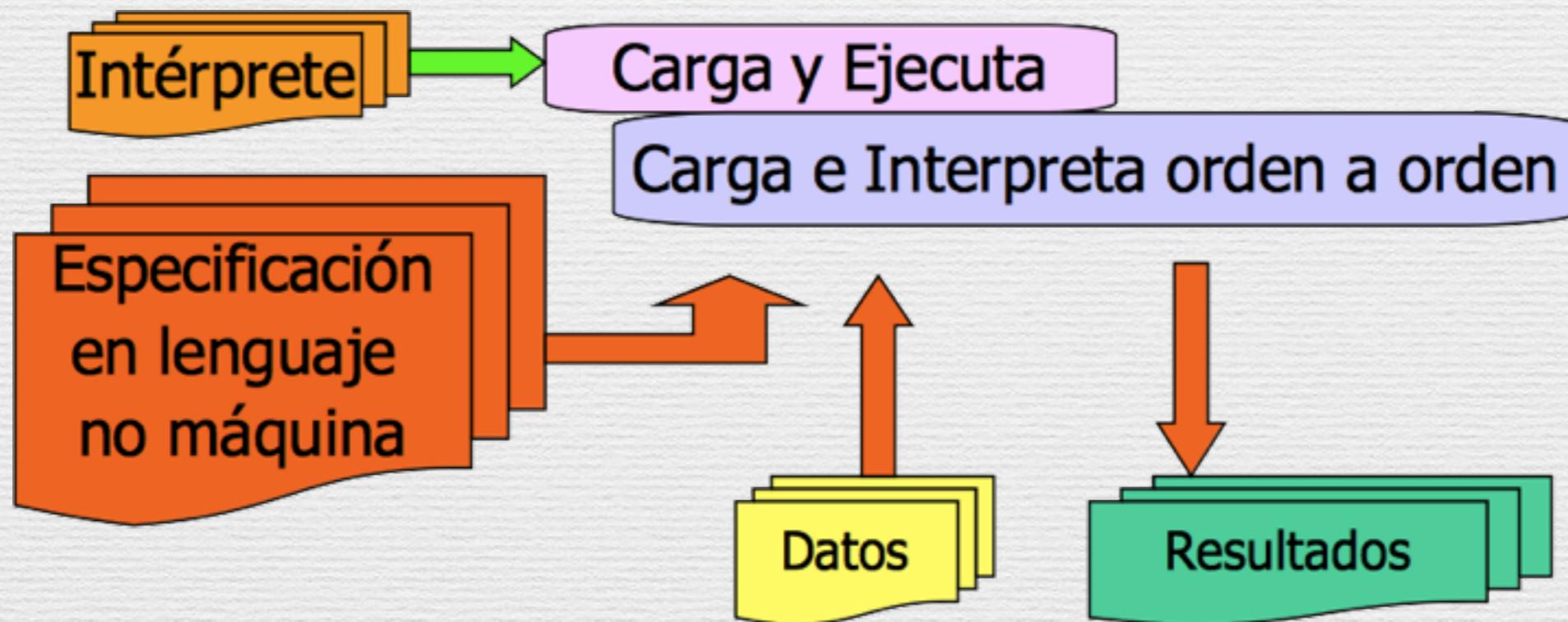
La compilación es un proceso “lento” que se realiza en dos fases:

- ♦ Análisis del fuente
- ♦ Síntesis del objeto

# Intérprete

Intérprete lee un programa fuente escrito para una máquina virtual realiza la traducción de manera interna. Cada instrucción es cargada y ejecutada una a una para la máquina real.

No se genera ningún programa objeto equivalente al descrito en el programa fuente.



# Intérprete

## ¿Cuándo es útil un intérprete?

- El programador trabaja en un entorno interactivo y se desean obtener los resultados de la ejecución de una instrucción antes de ejecutar la siguiente.
- El programador lo ejecuta escasas ocasiones y el tiempo de ejecución no es importante.
- Las instrucciones del lenguaje tiene una estructura simple y pueden ser analizadas fácilmente.
- Cada instrucción será ejecutada una sola vez.

## ¿Cuándo no es útil un intérprete?

- Si las instrucciones del lenguaje son complejas.
- Los programas van a trabajar en modo de producción y la velocidad es importante
- Las instrucciones serán ejecutadas con frecuencia.

## 3.2. Construcción de Traductores

### Diferencias entre un Compilador y un Intérprete

#### Compilador

Tiene como salida un único lenguaje objeto.

El rendimiento en la ejecución del programa compilado (la salida) es más rápido que interpretado.

La salida puede depender de la arquitectura.

No requiere del programa fuente porque el programa objeto es ejecutable. Oculta el fuente.

Los errores sintácticos y semánticos se detectan antes de la ejecución del programa objeto.

Tiene menos flexibilidad en el uso de la memoria para el programa objeto.

#### Intérprete

Tiene como salida instrucciones traducidas.

El rendimiento (del intérprete) se somete al rendimiento de analizar la traducción una a una.

Tiende a ser más portable e independiente de la arquitectura.

Se requiere del lenguaje fuente para su ejecución. Más seguro.

Se detectan los errores en la ejecución del programa. Permite detectar e informar mejor de los errores.

Es más flexible para que el programa pueda usar la memoria.

## 3.2. Construcción de Traductores

### Esquema de traducción

1. Se parte de un programa fuente L
2. El analizador sintáctico verifica lexico y sintaxis.  
Comprueba las reglas de producción (P) en función de G  
La complejidad de la verificación sintáctica depende del tipo de gramática que define el lenguaje.
3. El analizador semántico verifica validez de significado. Si es correcto tenemos un conjunto Inst\_L
4. El generador de código traduce: T(Inst\_L)
5. La salida es un programa objeto en L

# Gramáticas

Una *gramática libre de contexto*  $G$  es una 4-tupla  $(N, T, P, S)$ . Las sentencias de un lenguaje se dan mediante el conjunto de todas las producciones gramaticales. Se dice que  $L(G)$  es un lenguaje generado por una gramática.

Una gramática definida como  $G = (V_N, V_T, P, S)$ , donde:

- $V_N$  es el conjunto de símbolos no terminales.
- $V_T$  es el conjunto de símbolos terminales.
- $P$  es el conjunto de producciones.
- $S$  es el símbolo inicial.

[Aho08] (pp.197)

## 3.2. Construcción de Traductores

### Gramáticas: Ejemplo - Análisis Léxico

- P: 8 producciones
- No terminales: A, C, E, id, letra, dígito.
- Terminales: +, -, \*, /, =, ), (.
- Símbolo inicial: S.

$$\begin{array}{lcl} S & \rightarrow & A \mid C \\ A & \rightarrow & \text{id} = E \\ C & \rightarrow & \text{if } E \text{ then } S \\ E & \rightarrow & E \circ E \mid (E) \mid \text{id} \\ O & \rightarrow & + \mid - \mid * \mid / \\ \text{id} & \rightarrow & \text{letra} \mid \text{id} \text{ dígito} \mid \text{id} \text{ letra} \\ \text{letra} & \rightarrow & a \mid b \mid \dots \mid z \\ \text{dígito} & \rightarrow & 0 \mid 1 \mid \dots \mid 9 \end{array}$$

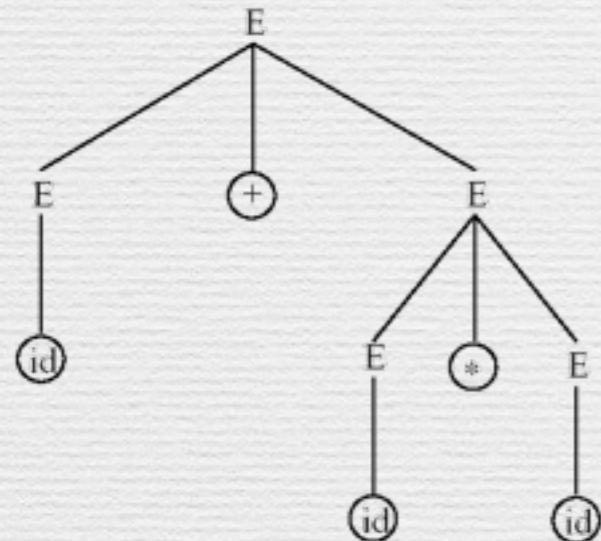
- ♦ Ejemplos:
  - ♦ Es válido: if a = 9 then bb3 = 8
  - ♦ No es válido: +a = o

## 3.2. Construcción de Traductores

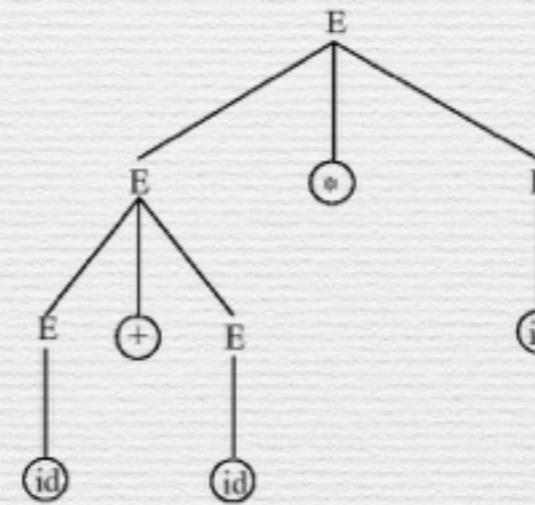
### Gramáticas ambiguas: Ejemplo - Análisis Sintáctico

Una gramática es ambigua cuando admite más de un árbol sintáctico para una misma secuencia de símbolos de entrada.

Dada secuencia de entrada **id+id\*id**, se puede apreciar que le pueden corresponder dos árboles sintácticos.



$$\begin{array}{l} E \rightarrow E + E \\ id + E \\ id + E * E \\ id + id * id \end{array}$$



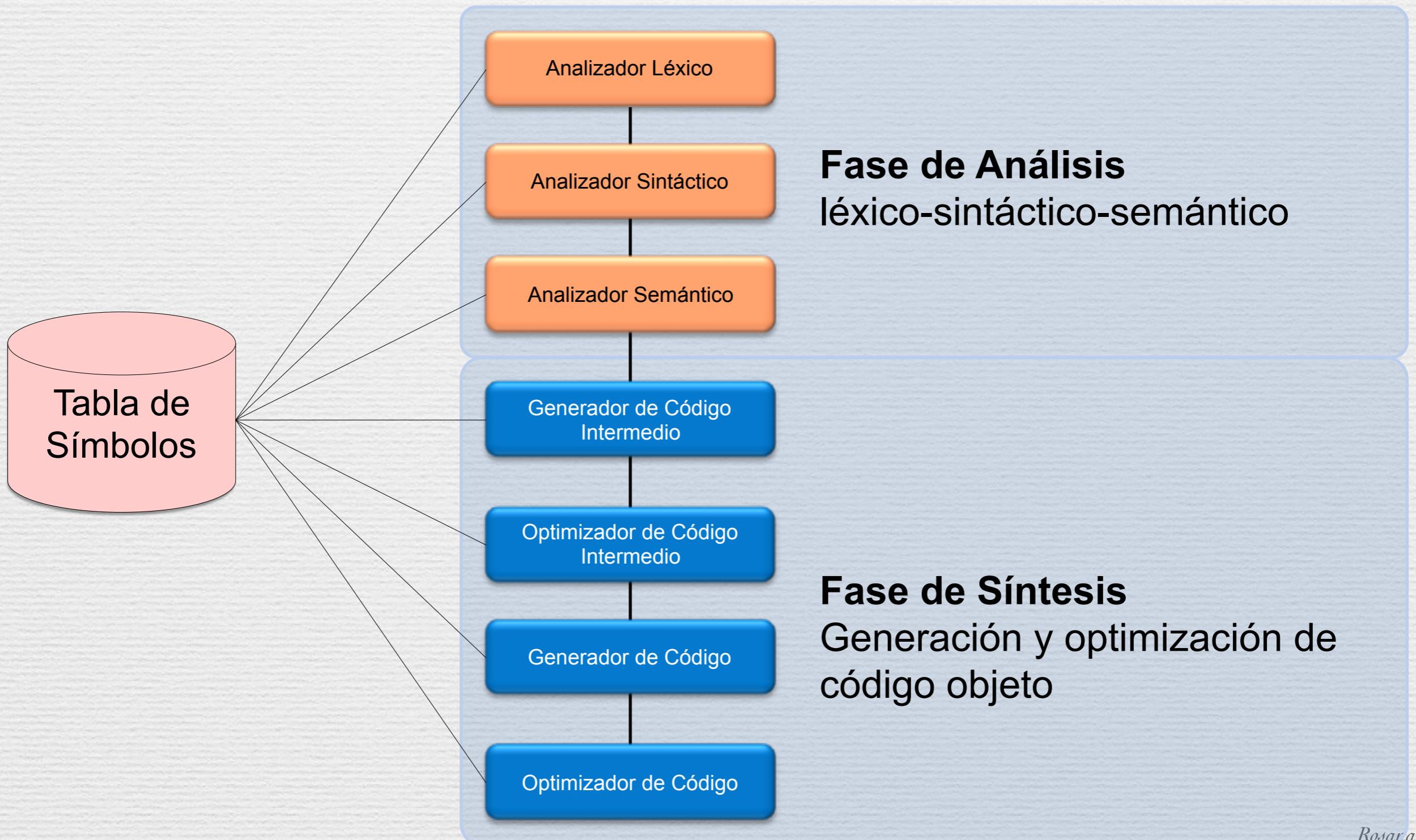
$$\begin{array}{l} E \rightarrow E * E \\ E * id \\ E + E * id \\ id + id * id \end{array}$$

Relacionado con:

- La precedencia de operadores.
- El uso de los paréntesis.

### 3.3. Fases de Traducción

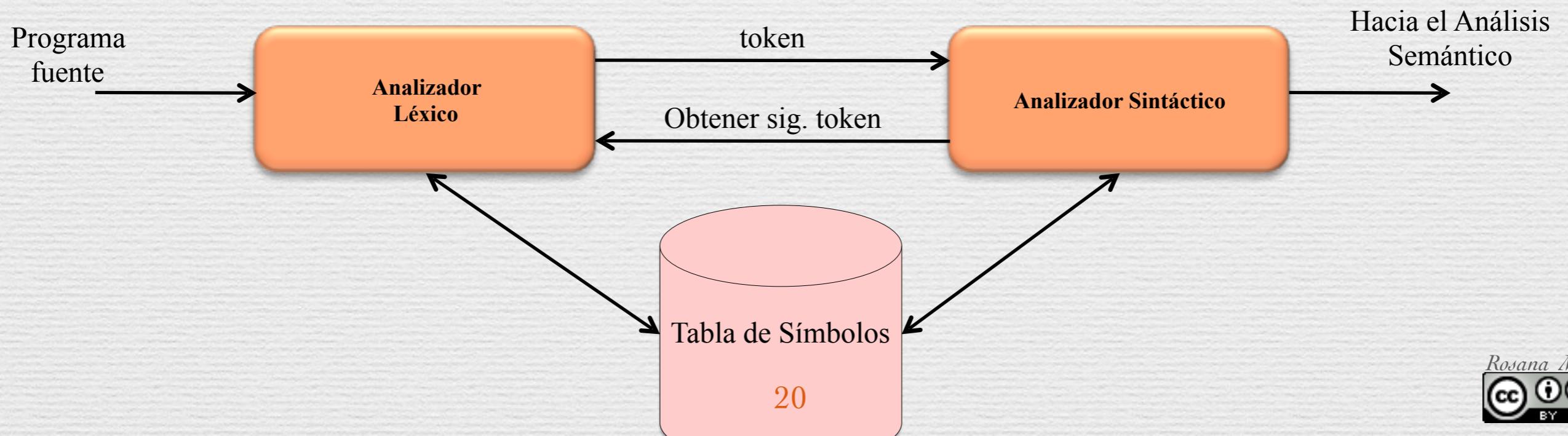
#### Fases en la construcción de un traductor



# Análisis Léxico

Función: procesar los caracteres de la entrada del programa fuente y producir como salida una secuencia de tokens.

- *Token*: Conjunto de lexemas con una misión sintáctica y tipo .
- *Lexema o Palabra*: Secuencia de caracteres del alfabeto con significado propio.
- *Patrón*: Especificación del fuente para el A.L., de la forma que se pueden tomar los lexemas de un token.



# Ejemplo 1 Análisis Léxico: lexemas

Token	Descripción informal	Lexemas de ejemplo
IF	<i>Caracteres ‘i’ y ‘f’</i>	<i>if</i>
ELSE	<i>Caracteres ‘e’, ‘l’, ‘s’ y ‘e’</i>	<i>else</i>
OP_COMP	<i>Operadores &lt;, &gt;, &lt;=, &gt;=, !=, ==</i>	<i>&lt;=, ==, !=, ...</i>
IDENT	<i>Letra seguida por letras y dígitos</i>	<i>pi, dato1, dato3, D3</i>
NUMERO	<i>Cualquier constante numérica</i>	<i>0, 210, 23.45, 0.899, ...</i>

- **Conjunto de lexemas**  
operadores, símbolos terminales, ...

# Análisis Léxico

Usualmente el lenguaje define los siguientes tokens:

- ♦ Un token para cada **palabra reservada** (if, do, while, else, ....).
- ♦ Los tokens para los **operadores** (individuales o agrupados).
- ♦ Un token que representa a todos los **identificadores** tanto de variables como de argumentos.
- ♦ Uno o más tokens que representan a las **constantes** (números y cadenas de literales).
- ♦ Tokens para cada **signo de puntuación** (paréntesis, llaves, coma, punto, punto y coma, corchetes, ....).

**Error léxico:** cuando el carácter de la entrada no tenga asociado a ninguno de los patrones disponibles en nuestra lista de tokens.

Ej: carácter extraño en la formación de una palabra reservada: `whi¿le`

### 3.3. Fases de Traducción

#### Especificación de los Tokens usando expr. regulares

Las *expresiones regulares* identifican un patrón de símbolos del alfabeto como pertenecientes a un token determinado.

Los elementos utilizados son:

1. Cero o mas veces, operador \*.
2. Uno o más veces, operador +:  $r^* = r^+|\lambda$ .
3. Cero o una vez, operador ?.
4. Una forma cómoda de definir clases de caracteres es de la siguiente forma:  
 $a|b|c|\dots|z = [a - z]$

## Ejemplo 2 Análisis Léxico: especificación de los Tokens

Dada la gramática o su diagrama de transiciones asociado, los patrones que van a definir a los tokens son:

S	$\rightarrow$	A   C
A	$\rightarrow$	id = E
C	$\rightarrow$	if E then S
E	$\rightarrow$	E O E   (E)   id
O	$\rightarrow$	+   -   *   /
id	$\rightarrow$	letra   id digito   id letra
letra	$\rightarrow$	a   b   ...   z
digito	$\rightarrow$	0   1   ...   9

Token	Patrón
<b>ID</b>	<i>letra (letra digito)*</i>
<b>ASIGN</b>	"="
<b>IF</b>	"if"
<b>THEN</b>	"then"
<b>PAR_IZQ</b>	"("
<b>PAR_DER</b>	
<b>OP_BIN</b>	"+   "-"   "*"   "/"

# Análisis Sintáctico

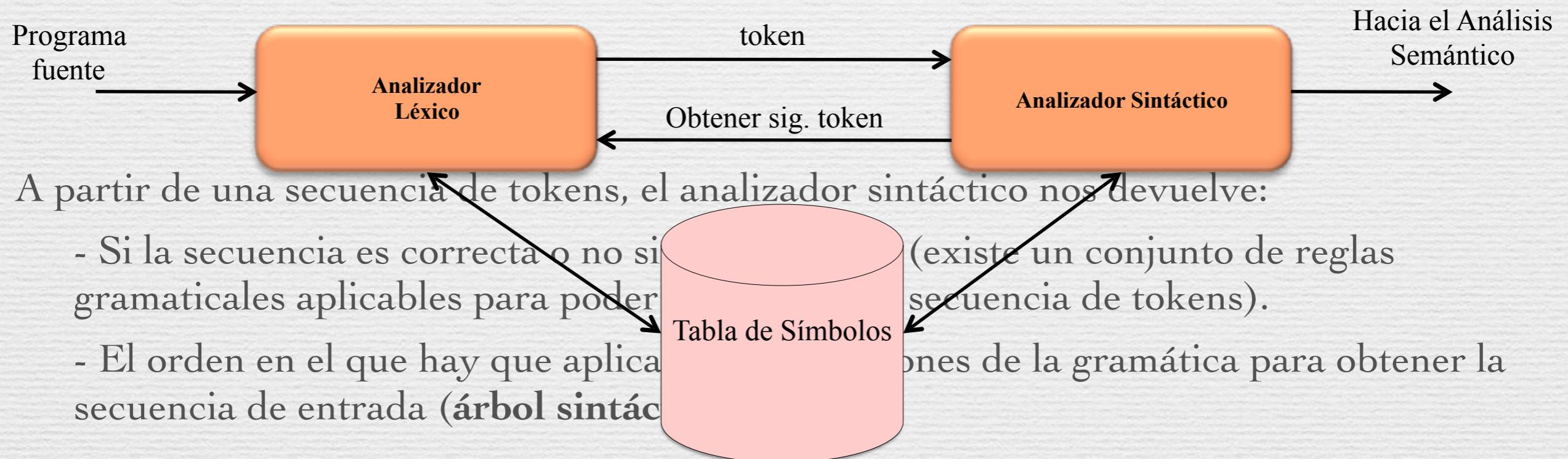
Las gramáticas ofrecen beneficios considerables tanto para los que diseñan lenguajes como para los que diseñan los traductores.

- Una gramática proporciona una especificación sintáctica precisa de un lenguaje de programación.
- A partir de ciertas clases gramaticales, es posible construir de manera automática un analizador sintáctico (YACC).
- Permite revelar ambigüedades sintácticas y puntos problemáticos en el diseño del lenguaje.
- Una gramática permite que el lenguaje pueda evolucionar o se desarrolle de forma iterativa agregando nuevas construcciones.

# Análisis Sintáctico

**Objetivo:** Analizar las secuencias de tokens y comprobar que son correctas sintácticamente.

Representa la estructura del programa fuente.



**Error sintáctico:** Si no se encuentra un árbol sintáctico para una secuencia de entrada, entonces la secuencia de entrada es incorrecta sintácticamente.

## Ejemplo 3 Análisis Sintáctico: verificación sintáctica

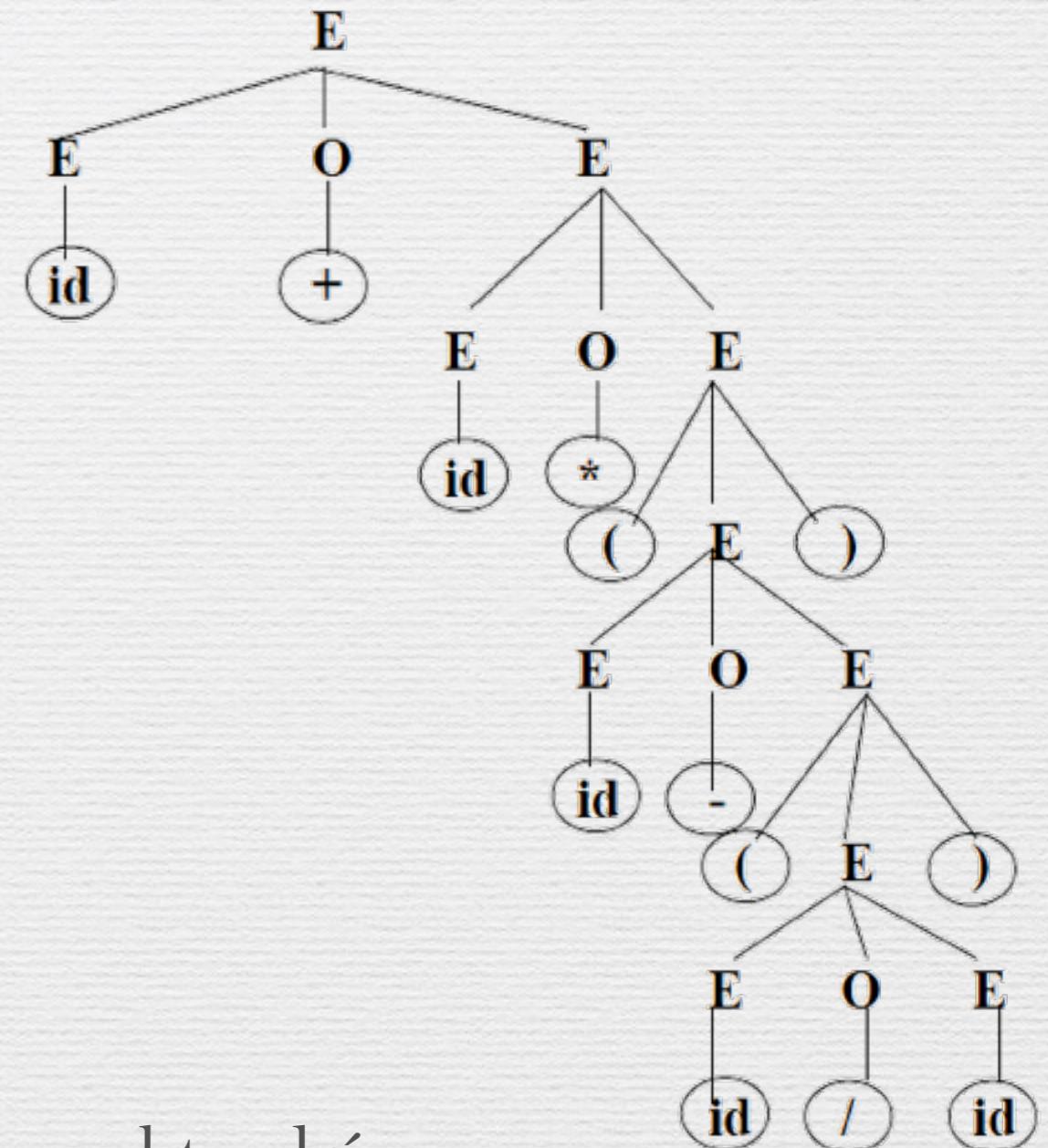
Dada la siguiente gramática G:

$$\begin{array}{lcl} P & = & \{ \quad E \rightarrow E O E \\ & & | \quad ( E ) \\ & & | \quad id \\ O & \rightarrow & + | - | * | / \\ \} \end{array}$$

$$V_N = \{E, O\}$$

$$V_T = \{(,), id, +, -, *, /\}$$

$$S = E$$

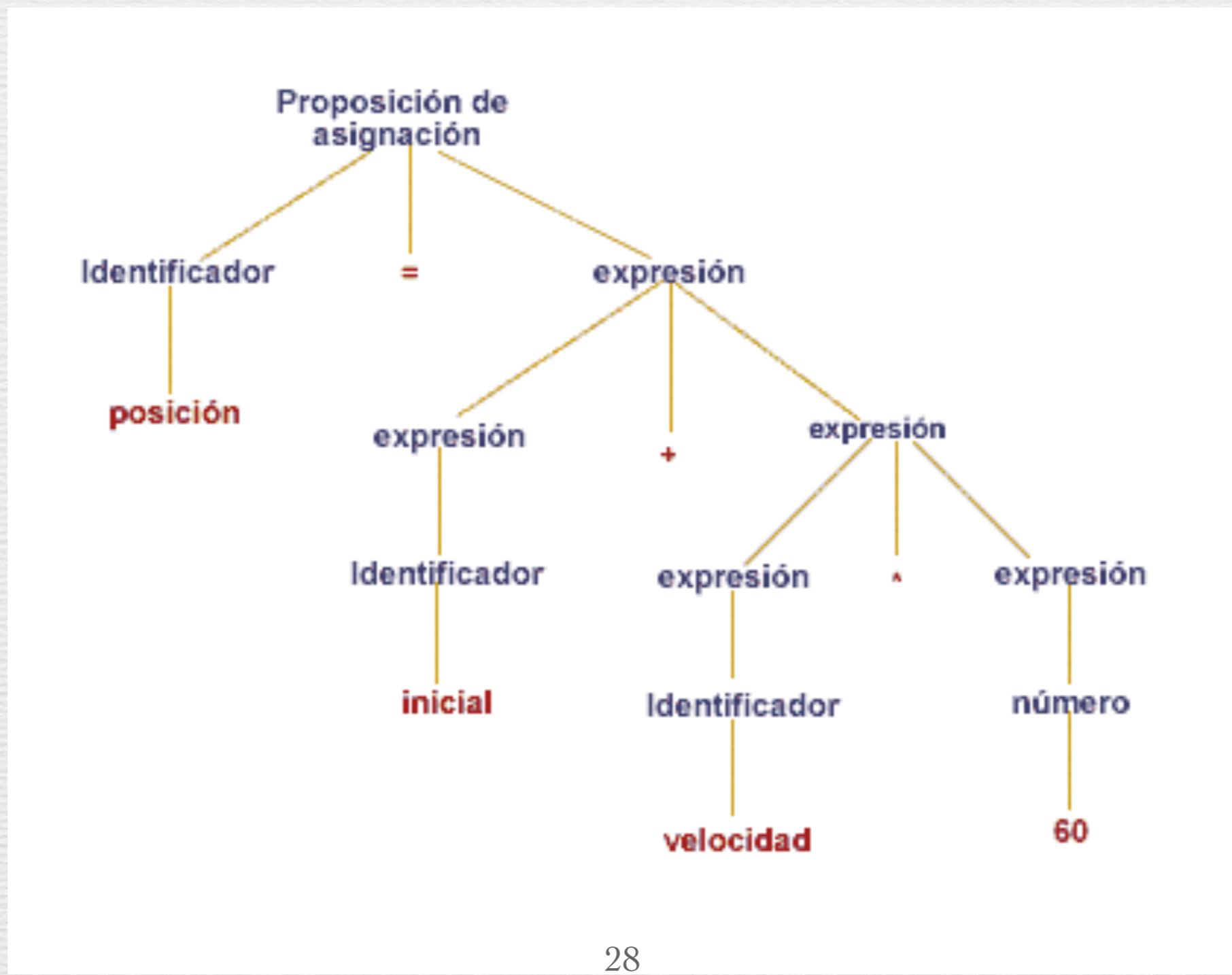


Y el texto de entrada: **id+id\*(id-(id/id))**

Usando las reglas de formación gramatical, se obtendría una representación que valida la construcción del texto de entrada -> verificación sintáctica correcta.

## Ejercicio: verificación sintáctica

Dado el siguiente árbol sintáctico, determine su gramática y el texto de entrada. Incluya cambios para que se produzca un error léxico y sintáctico.



# Análisis Semántico

**Semántica** de un **lenguaje de programación** es el significado dado a las distintas construcciones sintácticas.

El **significado** está ligado a la estructura sintáctica de las sentencias.

Asignación: según lenguaje C y la producción:

**sent\_asignacion** → **IDENTIFICADOR OP\_ASIG expresion PYC**

Tokens: **IDENTIFICADOR** (variable), **OP\_ASIG** (=) y **PYC** (;) son

Reglas semánticas:

- **IDENTIFICADOR** debe estar previamente declarado.
- El tipo de la expresión debe ser acorde al tipo del **IDENTIFICADOR**.

# Análisis Semántico

Durante la fase de análisis semántico se producen errores cuando se detectan construcciones **sin un significado correcto**:

- ♦ variable no declarada,
- ♦ tipos incompatibles en una asignación:
  - ♦ en C es posible realizar asignaciones entre variables de distintos tipos (Warnings).
  - ♦ Pascal lo impide tajantemente.
- ♦ llamada a un procedimiento incorrecto
- ♦ llamada con número de argumentos incorrectos,
- ♦ ...

**Error semántico:** cuando no se cumple una regla semántica.

Ej: el identificador no está previamente declarado: **da;ta**

# Generación de Código

- ♦ Generación de código intermedio:
  - ♦ Es más rápido construir compiladores para otros programas.
  - ♦ Entrada: árbol semántico.
  - ♦ Salida: código intermedio simple (casi lenguaje máquina)
- ♦ Optimización de código intermedio:
  - ♦ Solo optimizaciones independientes del lenguaje (varios saltos, bucles, expresiones comunes,...)
  - ♦ Entrada: código intermedio poco eficiente.
  - ♦ Salida: código intermedio mejorado.

# Generación de Código

## Ejemplo

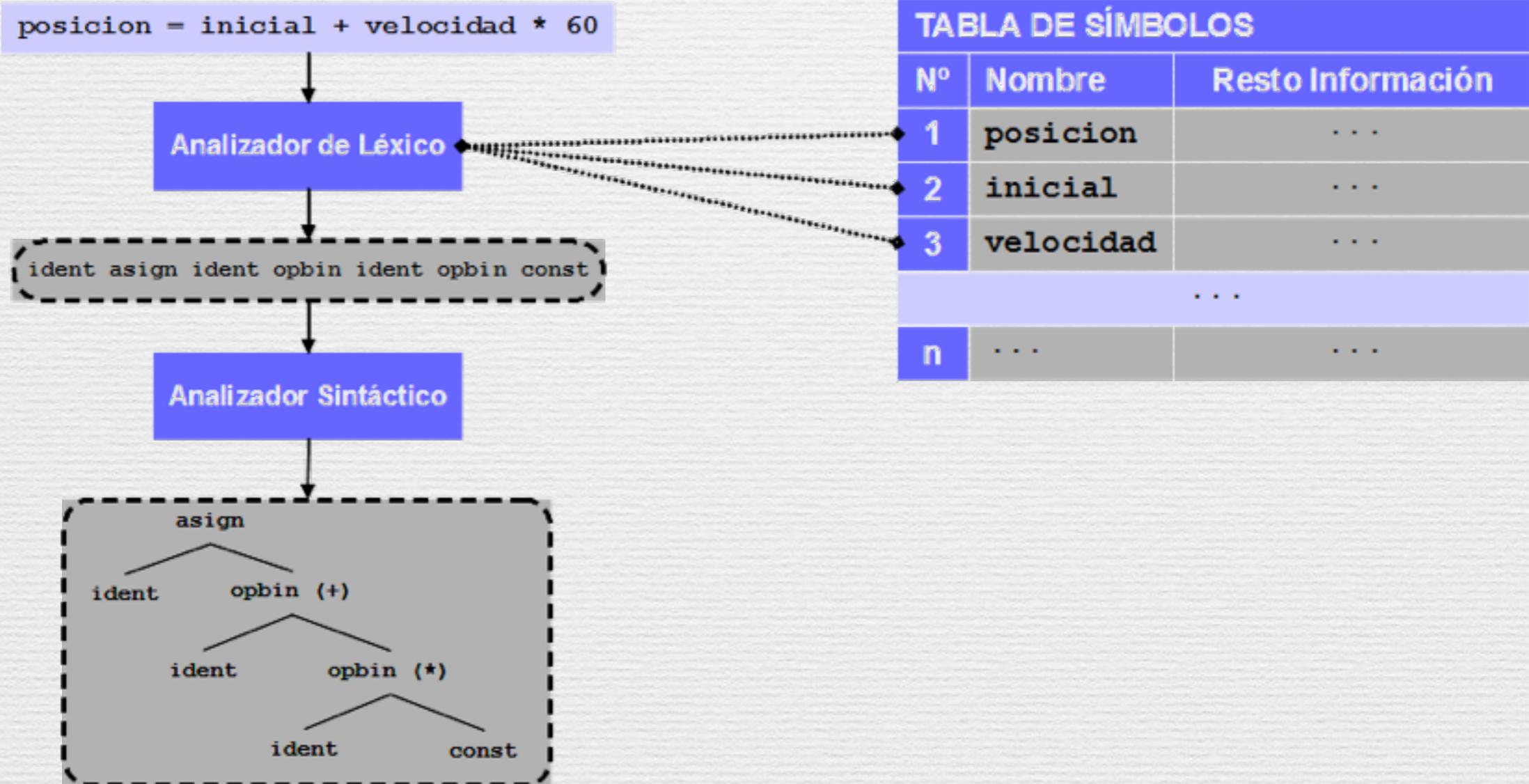
```
....  
for (i=0; i<1000; i++)  
{  
    r= 37.0-i*35;  
    b= 7.5; ←  
    z= b-sin(-r/35000);  
}
```

```
....  
b= 7.5;  
for (i=0; i<1000; i++)  
{  
    r= 37.0-i*35;  
    z= b-sin(-r/35000);  
}
```

- La última fase genera un archivo con un código en **lenguaje objeto** (generalmente lenguaje máquina) con el mismo significado que el texto fuente (traducción).
- Incluye la optimización del código objeto:
  - optimizaciones dependientes del hardware (compactar LOAD / STORE).

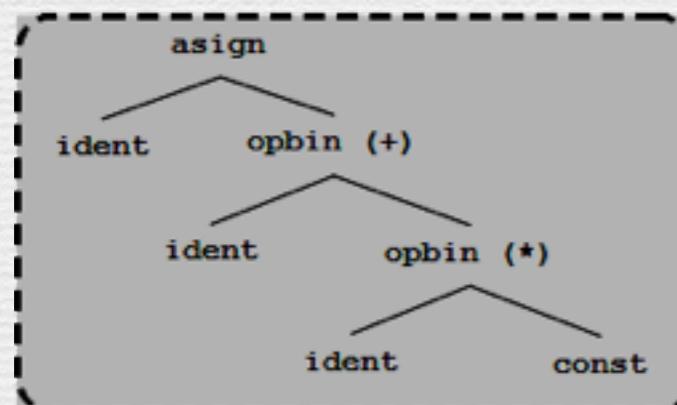
### 3.3. Fases de Traducción

#### Ejemplo (1/2) [Aho08]

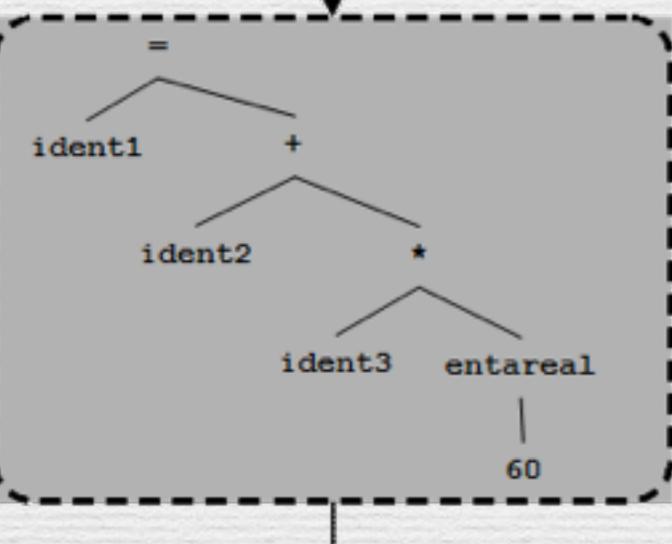


### 3.3. Fases de Traducción

#### Ejemplo (2/2) [Aho08]



Analizador Semántico



Generador de Código Intermedio

```
temp1 = entareal(60) ;  
temp2 = ident3 * temp1 ;  
temp3 = ident2 + temp2 ;  
ident1 = temp3 ;
```

Optimizador de Código

```
temp1 = ident3 * 60.0 ;  
Ident1 = ident2 + temp1 ;
```

Generador de Código

```
MOVF ident3, R2  
MULF #60.0, R2  
MOVF ident2, R1  
ADDF R2, R1  
MOVF R1, iden1
```

## 3.4. Modelos de Memoria de un Proceso

Sistemas Operativos Una visión aplicada [Carr07] (pp.219-231)

Elementos responsables de la gestión de memoria:

- ♦ Lenguaje de programación
- ♦ Compilador
- ♦ Enlazador
- ♦ Sistema operativo
- ♦ MMU – Memory Management Unit

## 3.4. Modelos de Memoria de un Proceso

### Niveles de la Gestión de Memoria

- **Nivel de procesos** - reparto de memoria entre los procesos.  
Responsabilidad del SO.
- **Nivel de regiones** - distribución del espacio asignado a un proceso a las regiones del mismo. Gestionado por el SO.
- **Nivel de zonas** - reparto de una región entre las diferentes zonas (nivel estático, dinámico basado en pila y dinámico basado en heap) de ésta. Gestión del lenguaje de programación con soporte del SO.

## **3.4. Modelos de Memoria de un Proceso**

### **Necesidades de Memoria de un Proceso**

- ♦ Tener un espacio lógico independiente.
- ♦ Espacio protegido del resto de procesos.
- ♦ Posibilidad de compartir memoria.
- ♦ Soporte a diferentes regiones.
- ♦ Facilidades de depuración.
- ♦ Uso de un mapa amplio de memoria.
- ♦ Uso de diferentes tipos de objetos de memoria.
- ♦ Persistencia de datos.
- ♦ Desarrollo modular.
- ♦ Carga dinámica de módulos.

## 3.4. Modelos de Memoria de un Proceso

**Sistemas Operativos Una visión aplicada [Carr07] (pp.246-251)**

La gestión del mapa de memoria de un proceso va desde la generación del ejecutable a su carga en memoria.

Operaciones:

- A nivel de proceso: creación, eliminación, duplicado.
- A nivel de región: creación, eliminación, duplicado.
- A nivel de zona: reserva, liberación, redimensión.

Relativo a los conceptos de:

- Tipos de objetos necesarios por un programa y su implementación.
- Ciclo de vida de un programa.
- Estructura de un ejecutable.
- Bibliotecas.

# Objetos de memoria

- ♦ Datos estáticos:
  - ♦ Existen durante toda la vida del programa.
  - ♦ Constantes vs variables
  - ♦ Con vs sin valor inicial (ver PIC)
- ♦ Datos dinámicos asociados a la ejecución de una función:
  - ♦ Se almacenan en pila: registro de activación, variables locales, parámetros de función, dirección de retorno.
- ♦ Datos dinámicos controlados por el programa – *heap*

# Ejemplos de Tipos de Objetos de Memoria

## Objetos de Memoria:

- ♦ variables / constantes
- ♦ globales / locales / externas
- ♦ con / sin valor inicial
- ♦ estáticas / dinámicas

```
int a;
int b= 8;
static int c;
static int d= 8;
const int e= 8;
static const int f= 8;
extern int g;

void mifuncion (int h)
{
    int i;
    int j= 8;
    static int k;
    static int l= 8;
    {
        int m;
        int n= 8;
    }
    . . .
}
```

## 3.4. Modelos de Memoria de un Proceso

### Ejemplo que usa los tres Tipos de Objetos de Memoria Básicos

```
struct punto {
    int x, y;
};

int main (int argc, char *argv[])
{
    static struct punto var_estatica;
    struct punto dinamica_pila;
    struct punto *dinamica_heap= malloc (sizeof (struct punto));

    var_estatica.y = 12;      /* 1 acceso con direccionamiento absoluto */
    dinamica_pila.y = 14;     /* 1 acceso con direccionamiento relativo a SP */
    dinamica_heap->y = 22;   /* 2 accesos con direccionamiento indirecto */

    return 0;
}
```

# Ejemplo de evolución de la Pila (Stack) en la ejecución de un programa

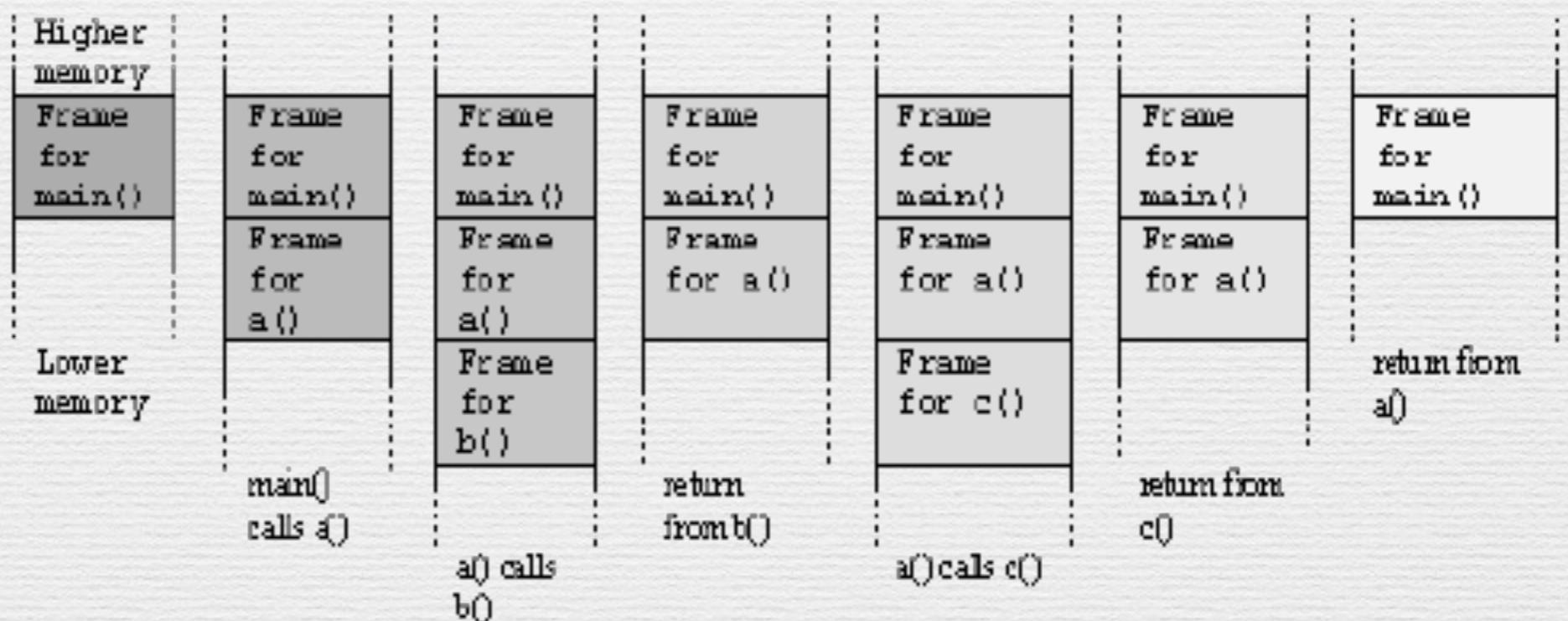
```
#include <stdio.h>
int a();
int b();
int c();
```

```
int a()
{
    b();
    c();
    return 0;
}
```

```
int b()
{
    return 0;
}
```

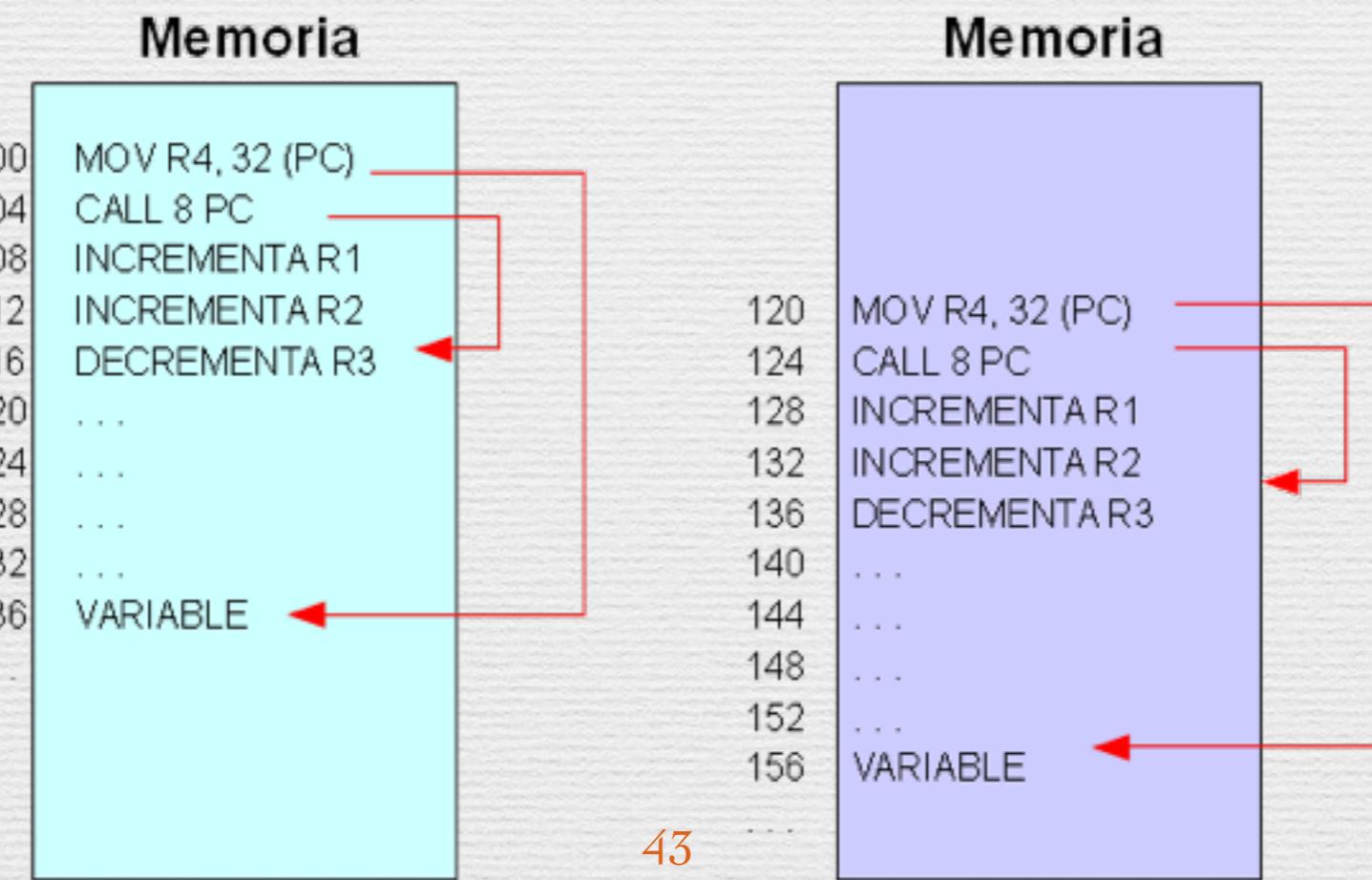
```
int c()
{
    return 0;
}
```

```
int main()
{
    a();
    return 0;
}
```



# Position Independent Code

- Un fragmento de código cumple esta propiedad si puede ejecutarse en cualquier parte de la memoria.
- Es necesario que todas sus referencias a instrucciones o datos no sean absolutas sino relativas a un registro, por ejemplo, contador de programa.



### 3.5. Ciclo de Vida de un Programa

[Carr07] (pp. 254-262)

A partir de un código fuente, un programa debe pasar por varias fases antes de poder ejecutarse:

1. Preprocesado
2. Compilación
3. Ensamblado
4. Enlazado
5. Carga y Ejecución



<http://slideplayer.es/slide/1715217/>

### 3.5. Ciclo de Vida de un Programa

El compilador (*compiler*):

- Genera código objeto y calcula cuánto espacio ocupan los diferentes tipos de datos.
- Asigna direcciones a los símbolos **estáticos** (instrucciones o datos) y resuelve las referencias bien de forma absoluta o relativa.
- Las referencias a símbolos **dinámicos** se resuelven usando direcciónamiento relativo a pila para datos relacionados a la invocación de una función, o con direcciónamiento indirecto para el heap. No necesitan reubicación al no aparecer en el archivo objeto.
- Genera la Tabla de Símbolos e información de depuración (si se solicita).

# 3.5. Ciclo de Vida de un Programa

## Ejemplo de Compilación

gcc o g++ asumen varias tareas:

```
$ gcc -v ejemplo.c
cpp1 ...           // preprocesador
cc ...             // compilador
as ...             // ensamblador
collect2 ...        // wrapper que invoca al enlazador ld
```

Podemos salvar los archivos temporales con:

```
$ gcc -f-save-temps
```

Podemos generar el archivo ensamblador con:

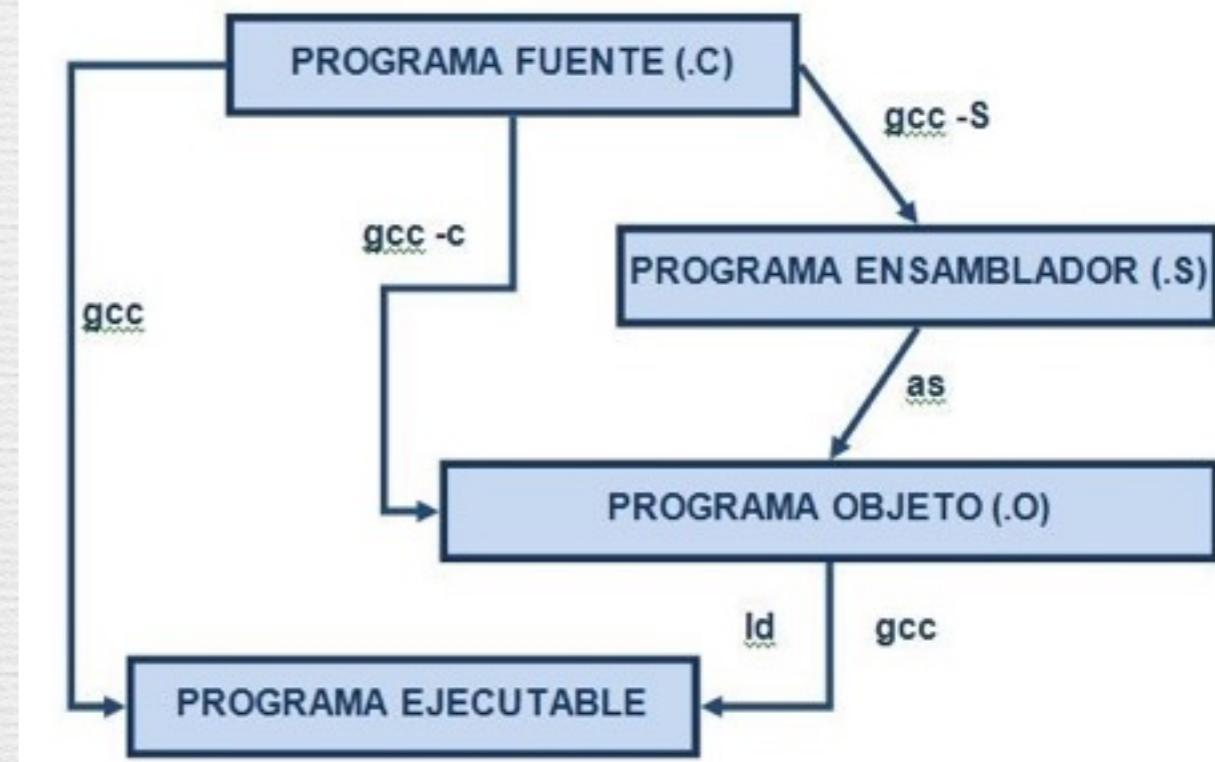
```
$ gcc -S
```

El archivo objeto es generado con el modificador:

```
$ gcc -c
```

Enlazar los objetos para generar el ejecutable con:

```
ld objeto.o -o eje
```



# 3.5. Ciclo de Vida de un Programa

## Ejemplo de Tabla de símbolos

```
#include <stdio.h>
int x = 42;

int main()
{
    printf("Hola Mundo, x = %d\n", x);
}
```

Tabla de símbolos:

```
$ gcc -c hola.c
$ nm hola.o
00000000 T main
                  U printf
00000000 D x
```

### 3.5. Ciclo de Vida de un Programa

El **enlazador** (*linker*) debe agrupar los archivos objetos de la aplicación y las bibliotecas, y resolver las referencias estáticas entre ellos.

Puede realizar reubicaciones dependiendo del esquema de gestión de memoria utilizado.

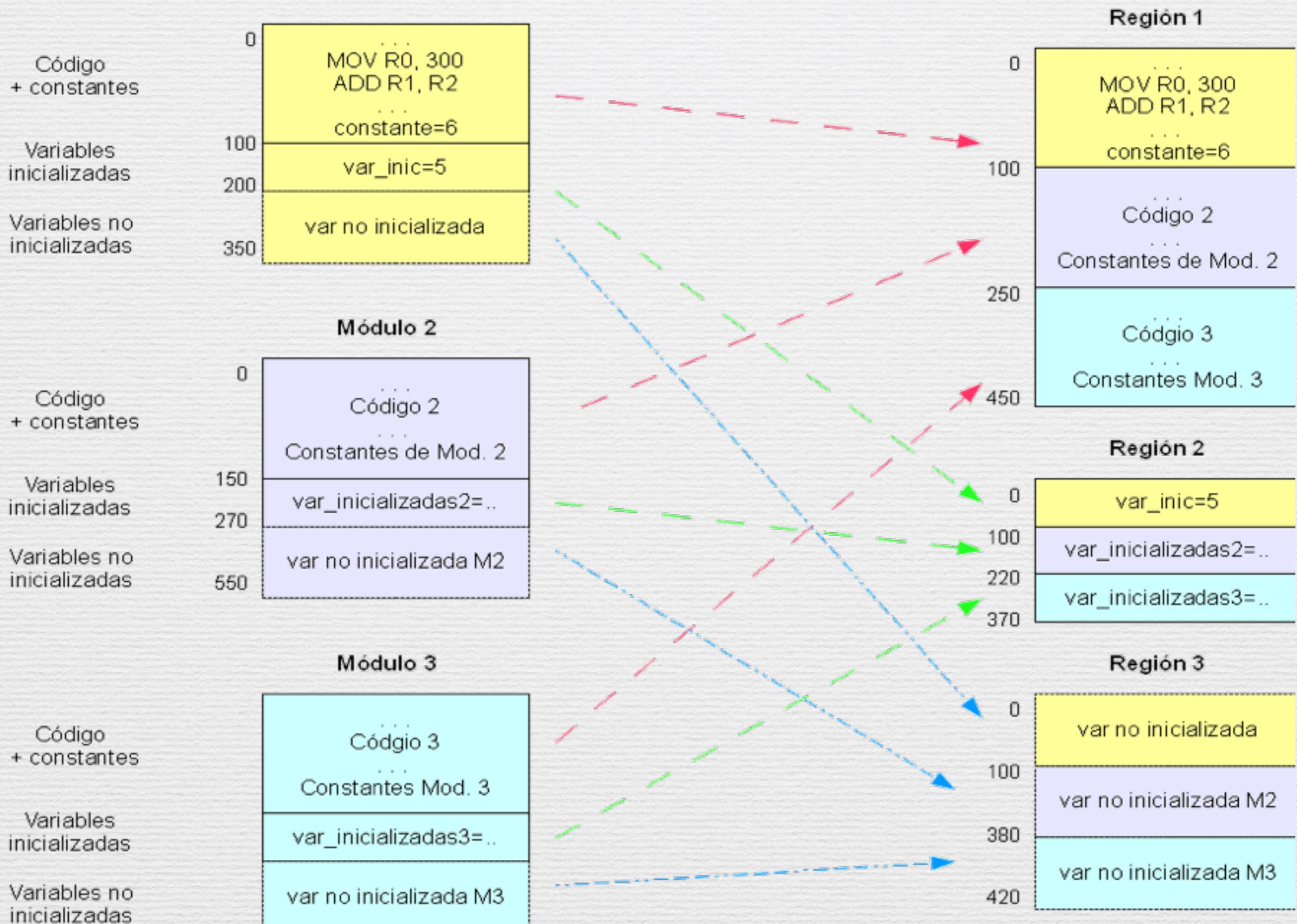
Segmentación	Paginación	Sin MMU
reubicación modulos	reub. modulos	reub. modulos
X	reub. regiones	reub. regiones

## 3.5. Ciclo de Vida de un Programa

### Funciones del Enlazador

- ♦ Resuelve los símbolos externos utilizando la tabla de símbolos.
- ♦ Agrupa las regiones de similares de los diferentes módulos en regiones (código, datos inicializados o no, etc.)
- ♦ Realiza la reubicación de módulos – transforma las referencias dentro de un módulo a referencias dentro de las regiones.
- ♦ Realiza la reubicación de regiones - transforma direcciones de una región en direcciones del mapa del proceso.

# Agrupamiento de módulos en regiones



### **3.5. Ciclo de Vida de un Programa**

#### **Carga y Ejecución**

- La reubicación del proceso se realiza en la carga o ejecución. Tres tipos, según el esquema de gestión de memoria:
  1. El cargador copia el programa en memoria sin modificarlo. Es la MMU la encargada de realizar la reubicación en tiempo de ejecución.
  2. En paginación, el hardware es capaz de reubicar los procesos en ejecución por lo que el cargador lo carga sin modificación.
  3. Si no usamos hardware de reubicación, ésta se realiza en la carga.

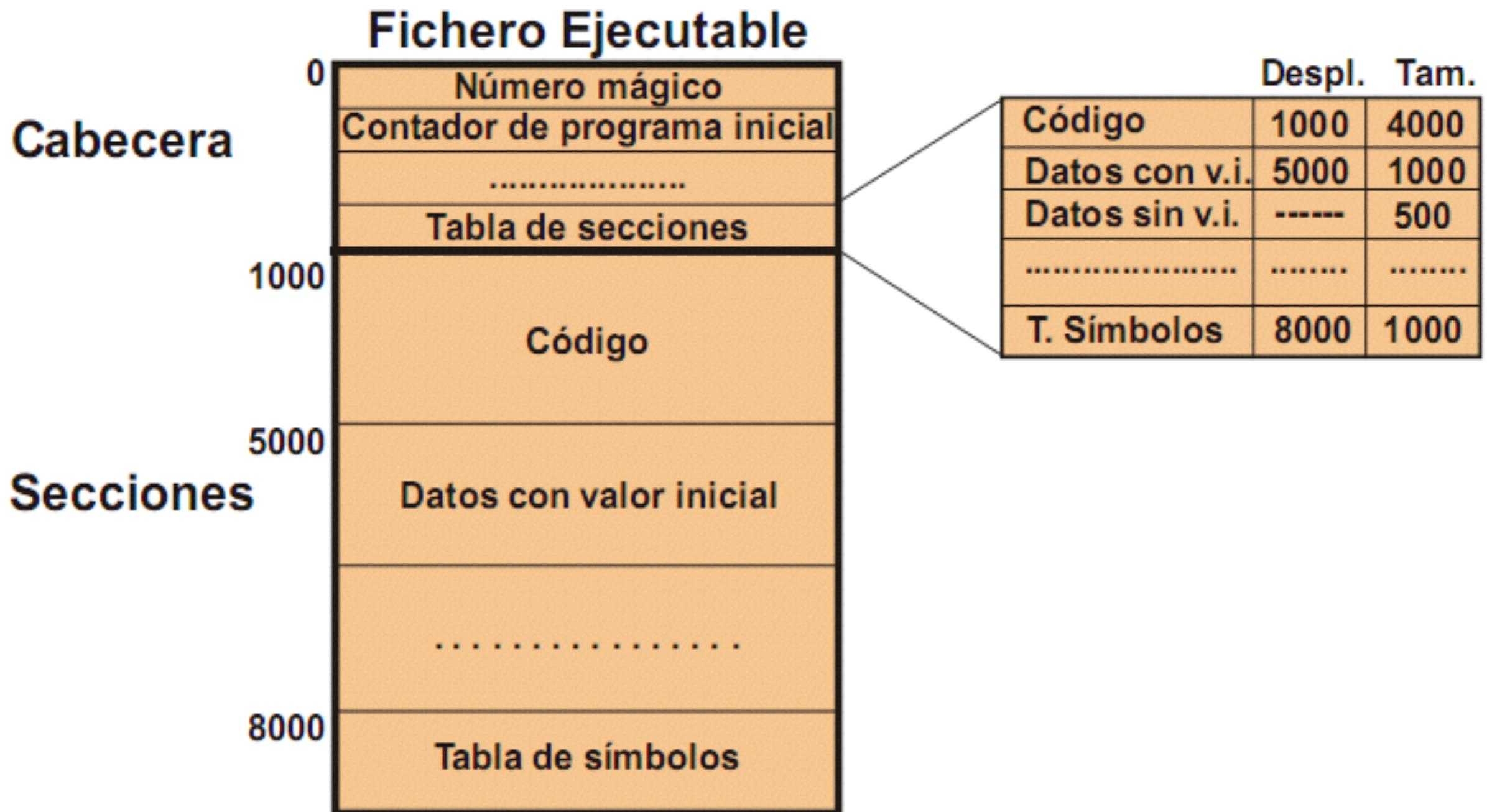
## 3.5. Ciclo de Vida de un Programa

### Diferencias entre archivos objeto y archivos ejecutables

- Los archivos **objeto** (resultado de la compilación) y **ejecutable** (resultado del enlazado) son muy similares en cuanto a contenidos.
- Su principales diferencias son:
  - En el ejecutable la cabecera del archivo contiene el punto de inicio del mismo, es decir, la primera instrucción que se cargará en el PC.
  - En cuanto a las regiones, sólo hay información de reubicación si ésta se ha de realizar en la carga.

### 3.5. Ciclo de Vida de un Programa

#### Formato de archivo ejecutable



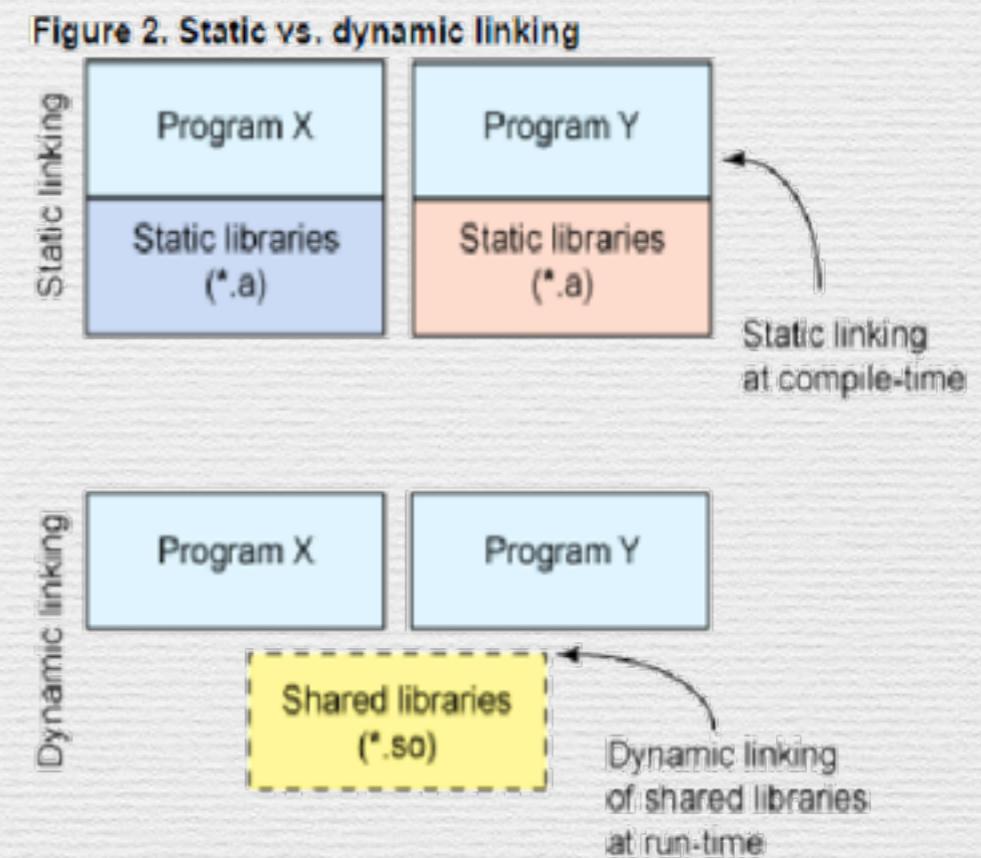
## Secciones de un archivo

- ♦ .text – *Instrucciones*. Compartida por todos los procesos que ejecutan el mismo binario. Permisos: r y w. Es de las regiones más afectada por la optimización realizada por parte del compilador.
- ♦ .bss – *Block Started by Symbol*: datos no inicializados y variables estáticas. El archivo objeto almacena su tamaño pero no los bytes necesarios para su contenido.
- ♦ .data – *Variables* globales y estáticas inicializadas. Permisos: r y w
- ♦ .rdata – *Constantes* o cadenas literales
- ♦ .reloc – Información de *reubicación para la carga*.
- ♦ **Tabla de símbolos** – Información necesaria (nombre y dirección) para localizar y reubicar definiciones y referencias simbólicas del programa. Cada entrada representa un símbolo.
- ♦ **Registros de reubicación** – información utilizada por el enlazador para ajustar los contenidos de las secciones a reubicar.

## 3.6. Bibliotecas

Sistemas Operativos Una visión aplicada [Carr07] (pp.262-267)

- ♦ **Biblioteca:** colección de objetos relacionados entre sí.
- ♦ Favorecen la modularidad y reusabilidad de código.
- ♦ Según su enlazado:
  - ♦ **Bibliotecas estáticas** - se enlazan con el programa en la compilación (.a)
  - ♦ **Bibliotecas dinámicas** – se enlazan en ejecución (.so)



# 3.6. Bibliotecas

## Creación y uso de Bibliotecas Estáticas

- ♦ Construimos el código fuente:

```
double media(double a, double b)
{
    return (a+b) / 2;
}
```

- ♦ Generamos el objeto:

```
gcc -c calc_mean.c -o calc_mean.o
```

- ♦ Archivamos el objeto (Creamos la biblioteca):

```
ar rcs libmean.a calc_mean.o
```

- ♦ Utilizamos la biblioteca:

```
gcc -static prueba.c -L. -lmean -o statically_linked
```

# 3.6. Bibliotecas

## Creación y uso de Bibliotecas Dinámicas

- Generamos el objeto de la biblioteca:

```
gcc -c -fPIC calc_mean.c -o calc_mean.o
```

- Creamos la biblioteca:

```
gcc -shared -Wl,-soname,libmean.so.1 -o libmean.so.1.0.1  
calc_mean.o
```

- Usamos la biblioteca:

```
gcc main.c -o dynamically_linked -L. -lmean
```

- Podemos consultar las bibliotecas enlazadas de un ejecutable:

```
ldd <ejecutable>
```

## 3.6. Bibliotecas

### Bibliotecas Dinámicas

Las bibliotecas estáticas tiene algunos inconvenientes:

- El código de la biblioteca esta en todos los ejecutables que la usan, lo que desperdicia disco y memoria.
- Si actualizamos las bibliotecas, debemos **recompilar** el programa para que se beneficie de la nueva versión.

Las bibliotecas dinámicas se integran en ejecución, para ello ya se ha realizado la reubicación de módulos.

Diferencia con un ejecutable: tienen tabla de símbolos, información de reubicación y no tiene punto de entrada.

Pueden ser:

- **Bibliotecas compartidas de carga dinámica** – la reubicación se realiza en tiempo de enlazado.
- **Bibliotecas compartidas enlazadas dinámicamente** – el enlazado se realiza en ejecución.

# Autoevaluación

## Entre teoría y prácticas yo puedo:

- ♦ Reconocer la diferencia entre dos o tres lenguajes de programación
- ♦ Diferenciar entre compilación e interpretación.
- ♦ Distinguir las fases que se producen en el proceso de traducción de un código fuente.
- ♦ Conozco la estructura de un ejecutable y cómo se establece la petición de
- ♦ Reconozco diferentes tipos de bibliotecas biblioteca estática.
- ♦ Trabajar de forma simple con editor, makefiles y depuradores en línea de comando.

Y además ...