

Tema 3. Compilación y Enlazado de Programas

Contenidos

- 3.1 Lenguajes de Programación.**
- 3.2 Construcción de Traductores.**
- 3.3 Proceso de Compilación.**
 - 3.3.1 Análisis Léxico.
 - 3.3.2 Análisis Sintáctico.
 - 3.3.3 Análisis Semántico.
 - 3.3.4 Generación y Optimización de Código.
- 3.4 Intérpretes.**
- 3.5 Modelos de Memoria de un Proceso.**
- 3.6 Ciclo de Vida de un Programa.**
- 3.7 Bibliotecas.**
- 3.8 Automatización del Proceso de Compilación y Enlazado.**

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.

Bibliografía básica

- [Prie06] A. Prieto, A. Lloris, J.C. Torres, **Introducción a la Informática (4ª Edición)**, McGraw-Hill, 2006
- [Carr07] J. Carretero, F. García, P. de Miguel, F. Pérez, **Sistemas Operativos (2ª Edición)**, McGraw-Hill, 2007
- [Aho08] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, **Compiladores. Principios, Técnicas y Herramientas (2ª Edición)**. Addison Wesley, 2008.

3.1 Lenguajes de Programación

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

Lenguaje de programación es un conjunto de símbolos y de reglas para combinarlos, que se usan para expresar algoritmos.

Características:

- Son **independientes** de la arquitectura física del computador, lo que aumenta la portabilidad de los programas.
- Una **instrucción** en un lenguaje de alto nivel da lugar, tras el proceso de traducción, a **varias instrucciones** en lenguaje máquina.
- Un lenguaje de alto nivel utiliza **notaciones fácilmente reconocibles** por las personas en el ámbito en que se usan.

| 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 |

Definición de Gramática

Una gramática proporciona una especificación sintáctica precisa de un lenguaje de programación.

La complejidad de la verificación sintáctica depende del tipo de gramática que define el lenguaje.

Una gramática se define 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 o reglas gramaticales.
- S es el símbolo inicial (es un símbolo no terminal).

3.2 Construcción de Traductores

Definición de Traductor

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.

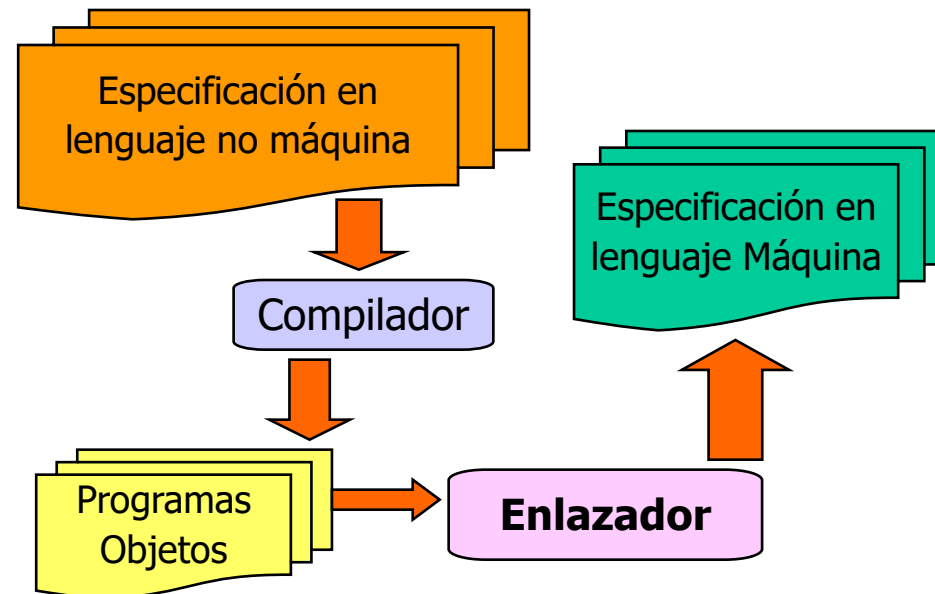
Existen dos tipos de traductores:

- **Compilador.**
- **Intérprete.**

Definición de Compilador

Compilador: traduce la especificación de entrada a lenguaje máquina incompleto y con instrucciones máquina incompletas -> se necesita un complemento llamado enlazador.

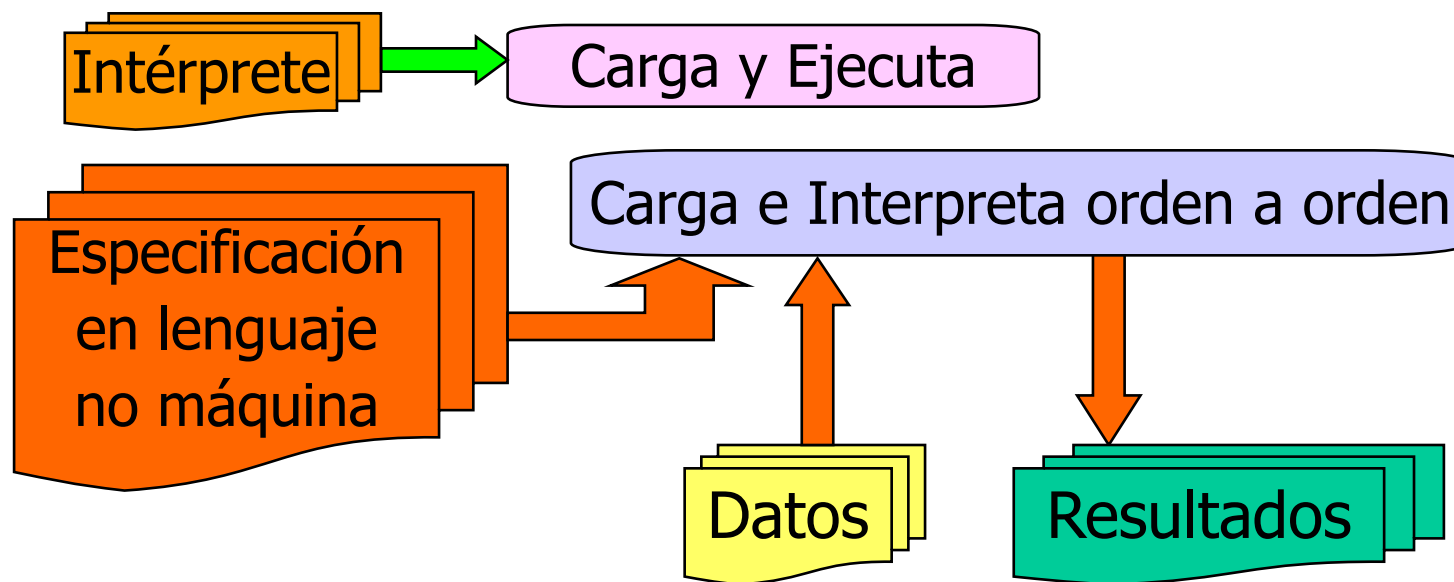
Enlazador (linker): completa los programas ligando las instrucciones máquina necesarias (añade rutinas binarias de funcionalidades no programadas directamente en el programa fuente) y genera un programa ejecutable para la máquina real.



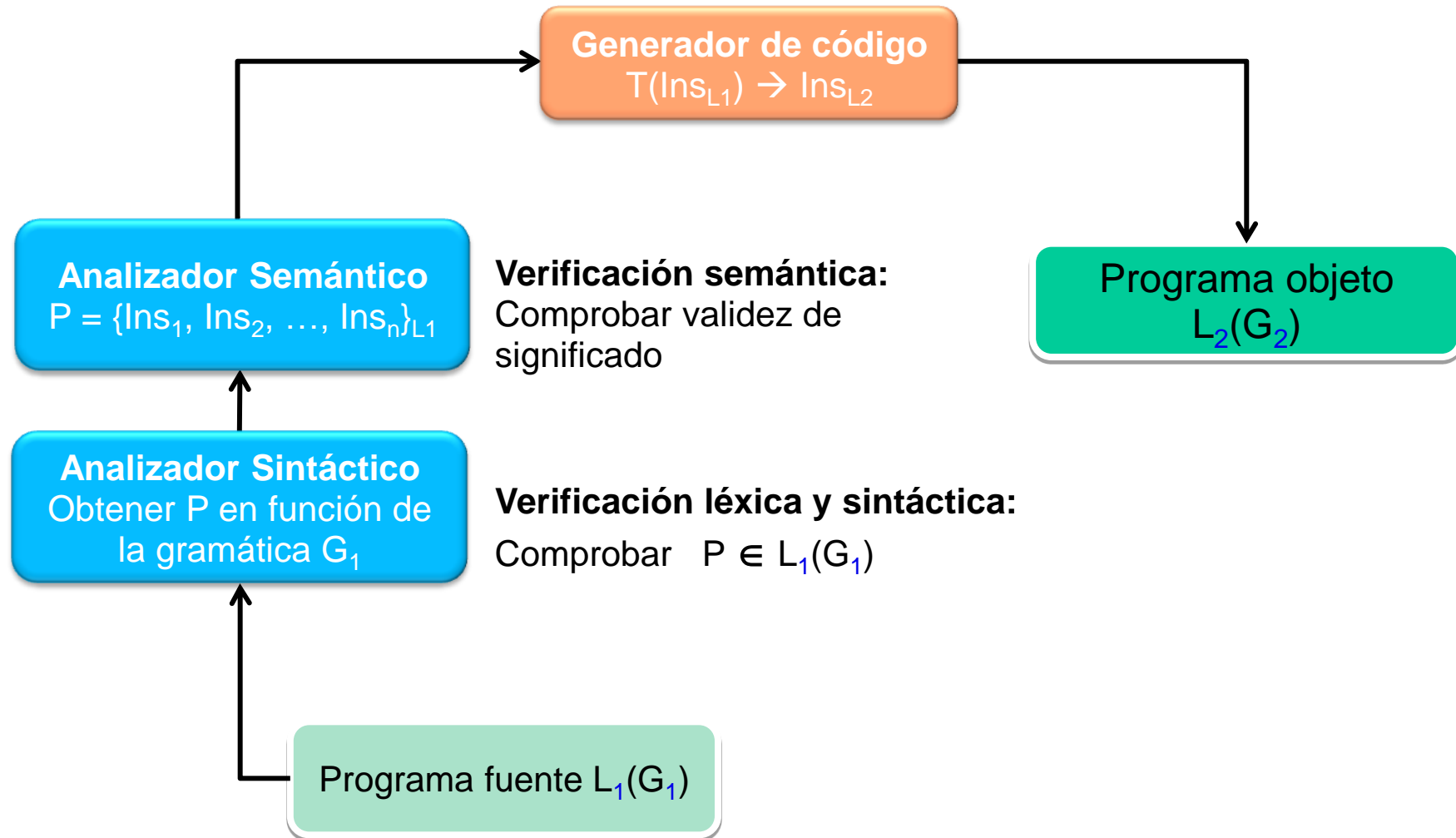
Definición de Intérprete

Intérprete: lee un programa fuente escrito para una máquina virtual, realiza la traducción de manera interna y ejecuta una a una las instrucciones obtenidas para la máquina real.

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



Esquema de Traducción



Definición de Gramática: conceptos previos [Aho08] (pp. 117-118, 197)

Alfabeto: conjunto finito de símbolos.

Ejemplo: Binario = { 0, 1 }

Cadena: secuencia finita de símbolos usando un alfabeto concreto.

Ejemplos: 110, 0110, 1, 00011

Símbolos terminales: elementos de un alfabeto usados para formar cadenas.

Ejemplos: 0, 1

Símbolos no terminales: variables sintácticas que representan conjuntos de cadenas (en el ejemplo, sería N). Se utilizan en las reglas gramaticales y no son elementos del alfabeto.

Ejemplo: $N \rightarrow N0 \mid N1 \mid 0 \mid 1$

Ejemplo

Dada la gramática siguiente:

$$P = \{ E \rightarrow E O E \mid (E) \mid id \\ O \rightarrow + \mid - \mid * \mid / \}$$

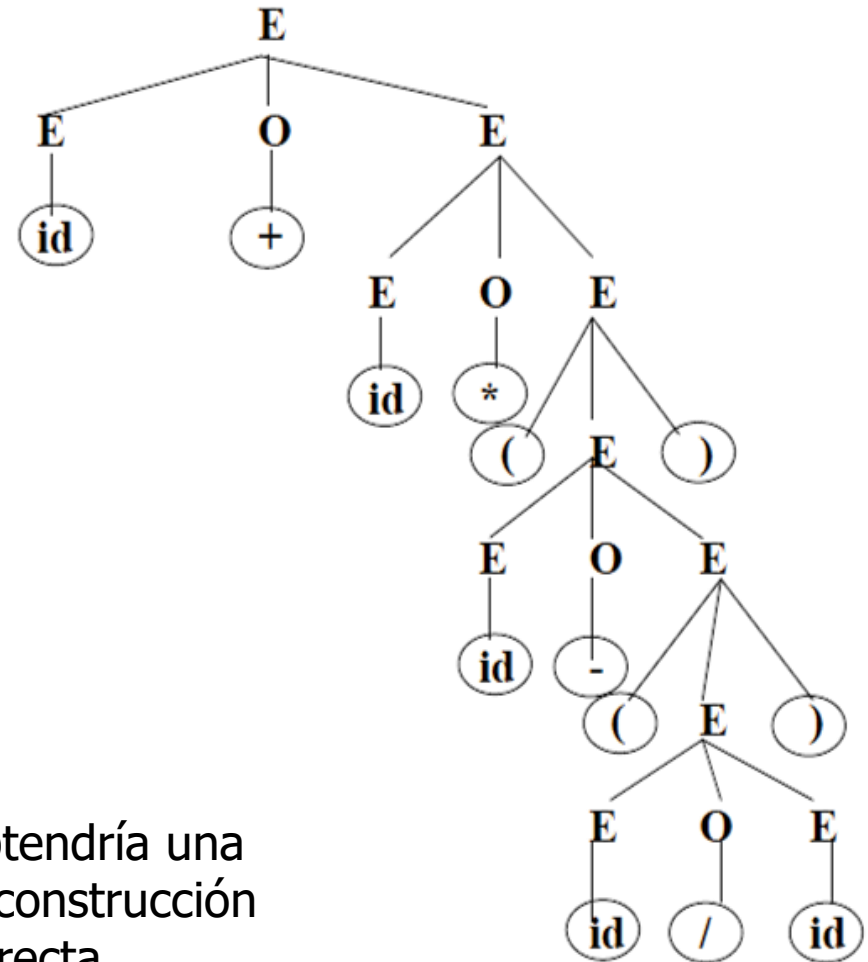
$$V_N = \{ E, O \}$$

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

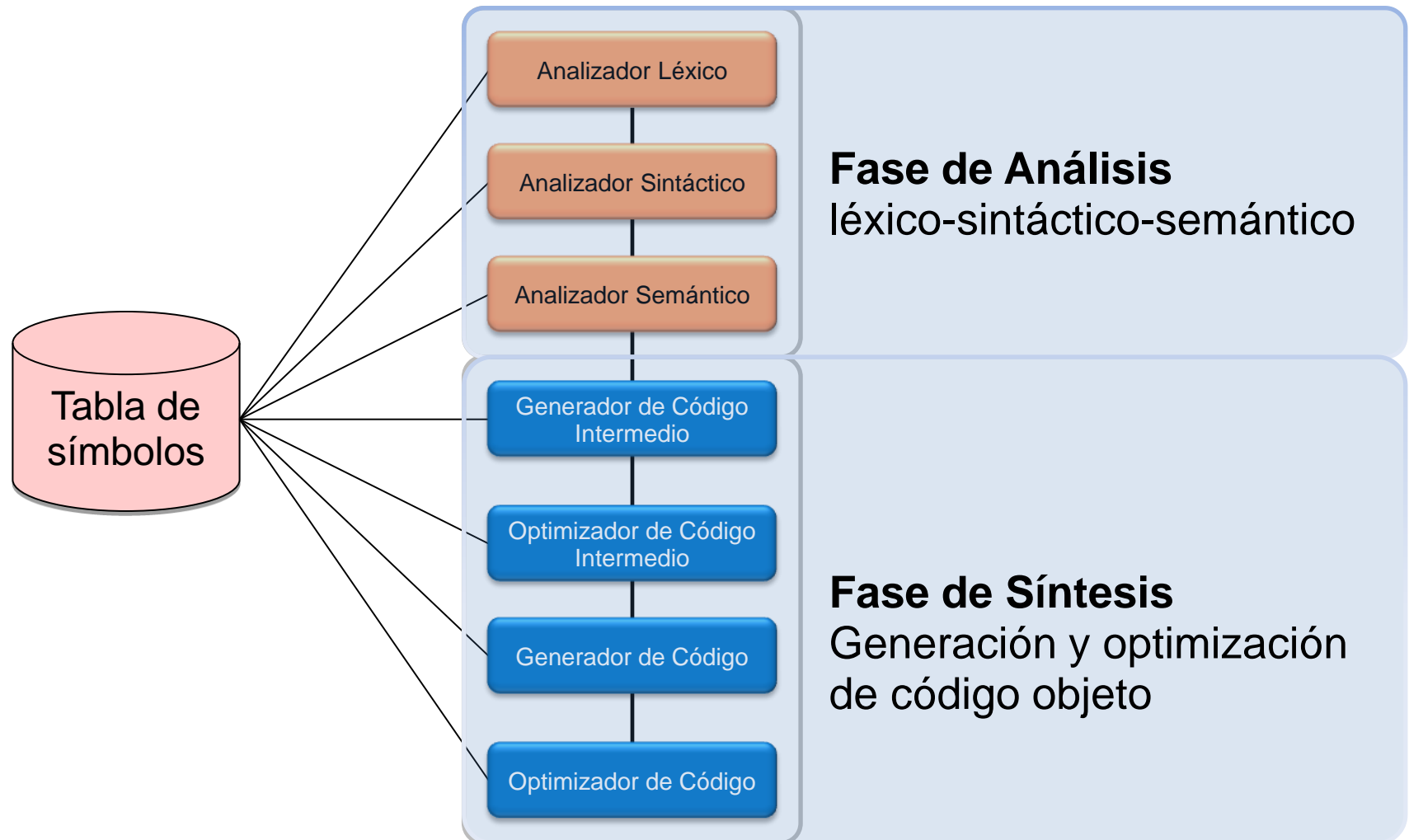
$$S = E$$

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

Usando las reglas de formación gramatical, se obtendría una representación (**árbol sintáctico**) que valida la construcción del texto de entrada \Rightarrow verificación sintáctica correcta.



Fases en la construcción de un traductor [Aho08] (pp. 4-11)

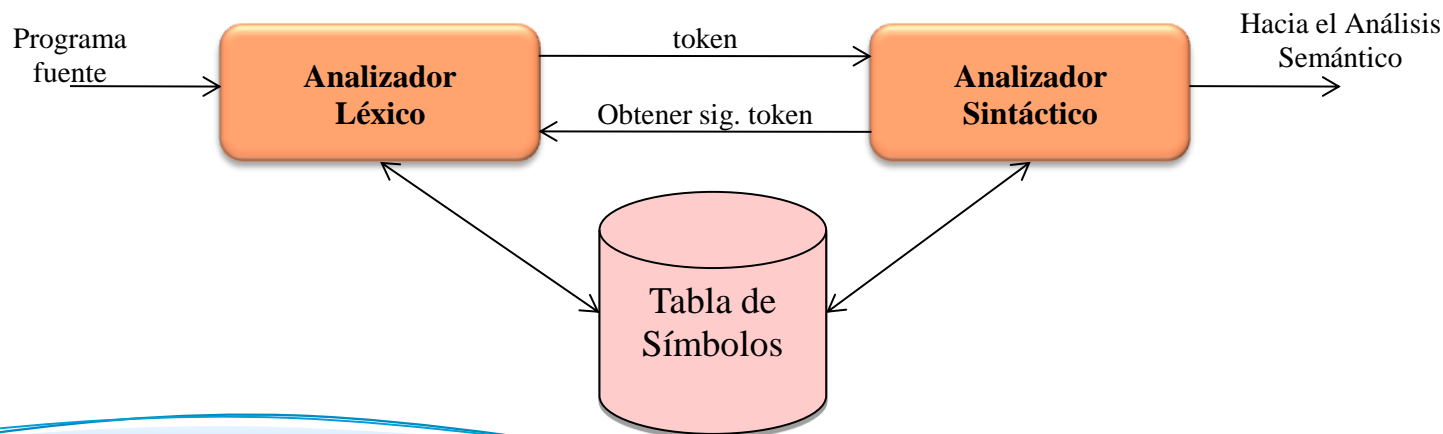


Análisis Léxico. Función Principal y Conceptos [Aho08] (pp. 109-111)

Función: Leer los caracteres de la entrada del programa fuente, agruparlos en lexemas (palabras) y producir como salida una secuencia de tokens para cada lexema en el programa fuente (eliminando caracteres superfluos y comentarios).

Conceptos que surgen del Analizador Léxico:

- **Lexema** o **Palabra**: Secuencia de caracteres del alfabeto con significado propio.
- **Token**: Concepto asociado a un conjunto de lexemas que, según la gramática del lenguaje fuente, tienen la misma misión sintáctica (en algunos casos, llevará asociado un valor de atributo).
- **Patrón**: Descripción de la forma que pueden tomar los lexemas de un token.



Análisis Léxico. Función Principal y Conceptos. Ejemplo

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

Análisis Léxico. Función Principal y Conceptos. Error Léxico

En muchos lenguajes de programación se consideran la mayoría de 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 subprogramas.
- Uno o más **tokens** que representan a las **constantes** (números y cadenas de literales).
- Un **token** para cada **signo de puntuación** (paréntesis izquierdo, paréntesis derecho, llave izquierda, llave derecha, coma, punto, punto y coma, corchete derecho, corchete izquierdo, ...).

Error léxico: Se producirá 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**).



Análisis Léxico. Especificación de los Tokens usando expresiones regulares

Se pueden usar expresiones regulares para identificar un patrón de símbolos del alfabeto como pertenecientes a un token determinado:

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:
 $a|b|c|\cdots|z = [a - z]$

Análisis Léxico. Especificación de los Tokens. Ejemplo

Dada la gramática mostrada anteriormente, los patrones que van a definir a los tokens serían los que muestra la tabla de la derecha:

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

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

Análisis Sintáctico

Las gramáticas ofrecen beneficios considerables tanto para los que diseñan lenguajes como para los que diseñan los traductores. Entre ellos destacamos los siguientes:

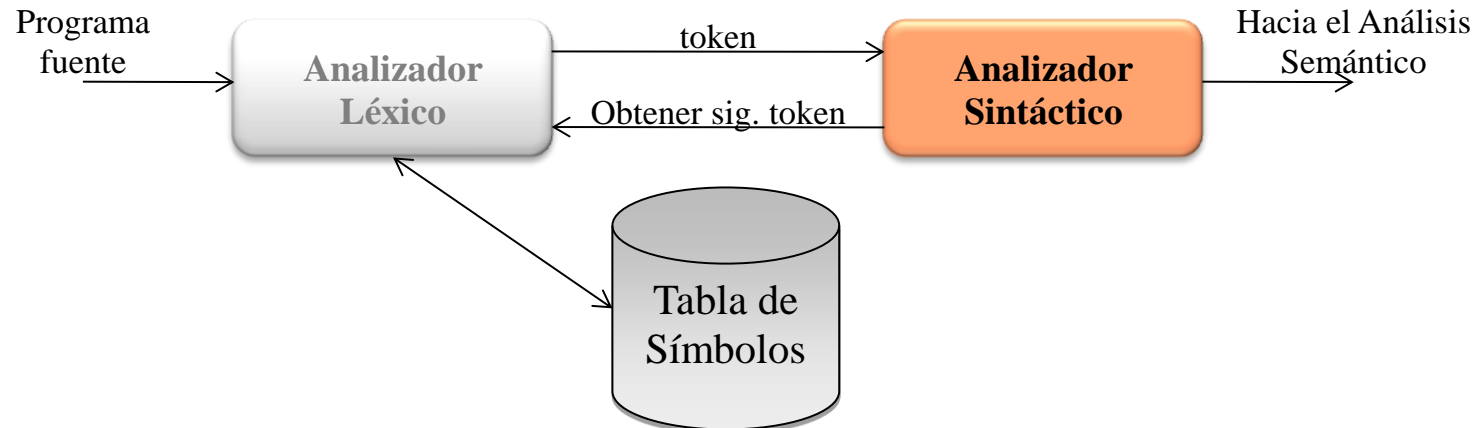
- Una gramática proporciona una especificación sintáctica precisa de un lenguaje de programación.
- A partir de ciertas clases gramaticales es posible construir de manera automática un analizador sintáctico eficiente.
- Permiten revelar ambigüedades sintácticas y puntos problemáticos en el diseño del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

Función del Analizador Sintáctico

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

A partir de una secuencia de tokens el analizador sintáctico nos devuelve:

- Si la secuencia es correcta o no sintácticamente (existe un conjunto de reglas gramaticales aplicables para poder estructurar la secuencia de tokens).
- El orden en el que hay que aplicar las producciones de la gramática para obtener la secuencia de entrada (**árbol sintáctico**).



Si no se encuentra un árbol sintáctico para una secuencia de entrada, entonces la secuencia de entrada es incorrecta sintácticamente (tiene errores sintácticos).

Análisis Sintáctico. Gramáticas Libres de Contexto [Aho08] (pp.197)

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

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

Se dice que es una **gramática libre de contexto** cuando el conjunto de producciones P es de la forma:

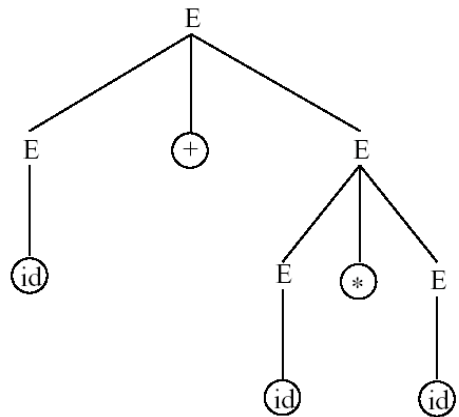
$$P = \{ A \rightarrow \alpha / A \in V_N, \alpha \in (V_N \cup V_T)^* \}$$

Es decir, solo admite tener un símbolo no terminal en la parte izquierda de las producciones. La denominación libre de contexto se debe a que donde aparezca A se podría poner α , independientemente del contexto en el que se encuentre A .

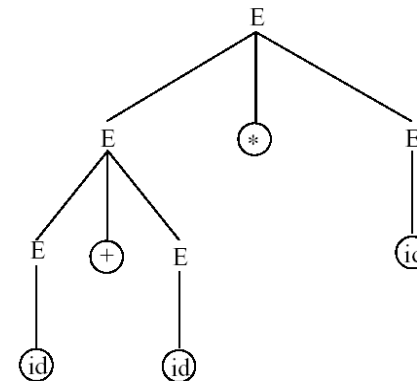
Análisis Sintáctico. Gramáticas ambiguas

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

Ejemplo: Dadas las producciones de la gramática del ejemplo de la diapositiva 9 y dada la misma secuencia de entrada **id+id*id**, se puede apreciar que le pueden corresponder dos árboles sintácticos.



$E \rightarrow E + E$
 $id + E$
 $id + E * E$
 $id + id * id$



$E \rightarrow E * E$
 $E * id$
 $E + E * id$
 $id + id * id$

Cuando programamos en un determinado lenguaje:

- ¿A qué nos referimos cuando hablamos de “**precedencia de operadores**”?
- ¿Por qué hay que utilizar los **paréntesis** para evitar la **precedencia de operador**?

Análisis Semántico

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

En los lenguajes de programación, el **significado** está ligado a la estructura sintáctica de las sentencias.

Ejemplo: En una sentencia de asignación, según la sintaxis del lenguaje C, expresada mediante la producción siguiente:

sent_asignacion → **IDENTIFICADOR** **OP_ASIG** **expresion** **PYC**

donde **IDENTIFICADOR**, **OP_ASIG** y **PYC** son símbolos terminales (tokens) que representan, respectivamente, a una variable, el operador de asignación "=" y al delimitador de sentencia ";", deben cumplirse las siguientes reglas semánticas:

- **IDENTIFICADOR** debe estar previamente declarado.
- El tipo de la **expresion** debe ser acorde con el tipo del **IDENTIFICADOR**.

3.3 Fases de Traducción

Análisis Semántico

- Durante la fase de análisis semántico se producen errores cuando se detectan construcciones **sin un significado correcto** (p.e. variable no declarada, tipos incompatibles en una asignación, llamada a un procedimiento incorrecto o con número de argumentos incorrectos, ...).
- En lenguaje C es posible realizar asignaciones entre variables de distintos tipos, aunque algunos compiladores devuelven **warnings** o avisos de que algo puede realizarse mal a posteriori.
- Otros lenguajes impiden la asignación de datos de diferente tipo (lenguaje Pascal).

3.3 Fases de Traducción

Generación de Código

- En esta fase se genera un archivo con un código en lenguaje objeto (generalmente lenguaje máquina) con el mismo significado que el texto fuente.
- En algunos, se intercala una fase de generación de código intermedio para proporcionar independencia de las fases de análisis con respecto al lenguaje máquina (portabilidad del compilador) o para hacer más fácil la optimización de código.

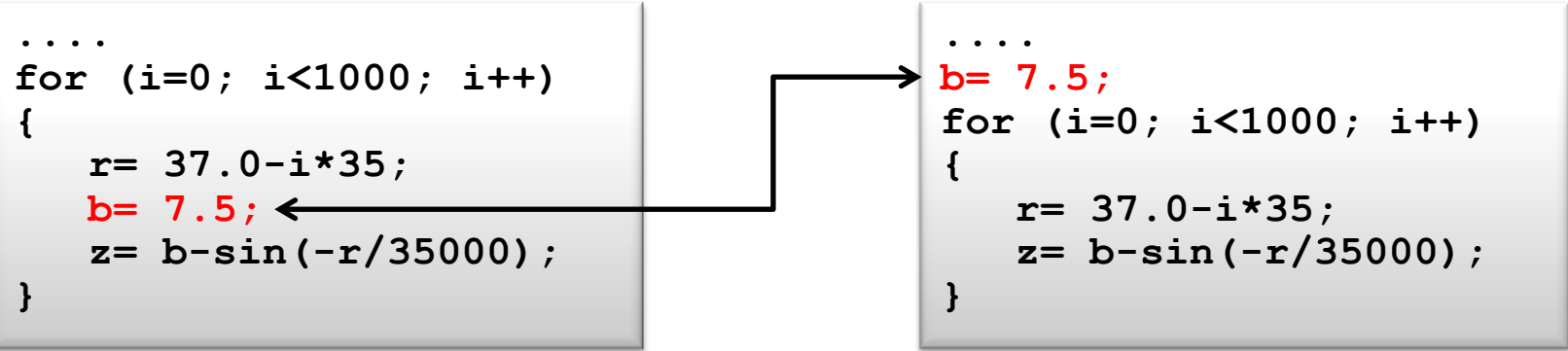
Optimización de Código

- Esta fase existe para mejorar el código mediante comprobaciones locales a un grupo de instrucciones (bloque básico) o a nivel global.
- Se pueden realizar optimizaciones de código tanto al código intermedio (si existe) como al código objeto final. Generalmente, las optimizaciones se aplican a códigos intermedios.

Ejemplo: Una asignación dentro de un bucle `for` en lenguaje C:

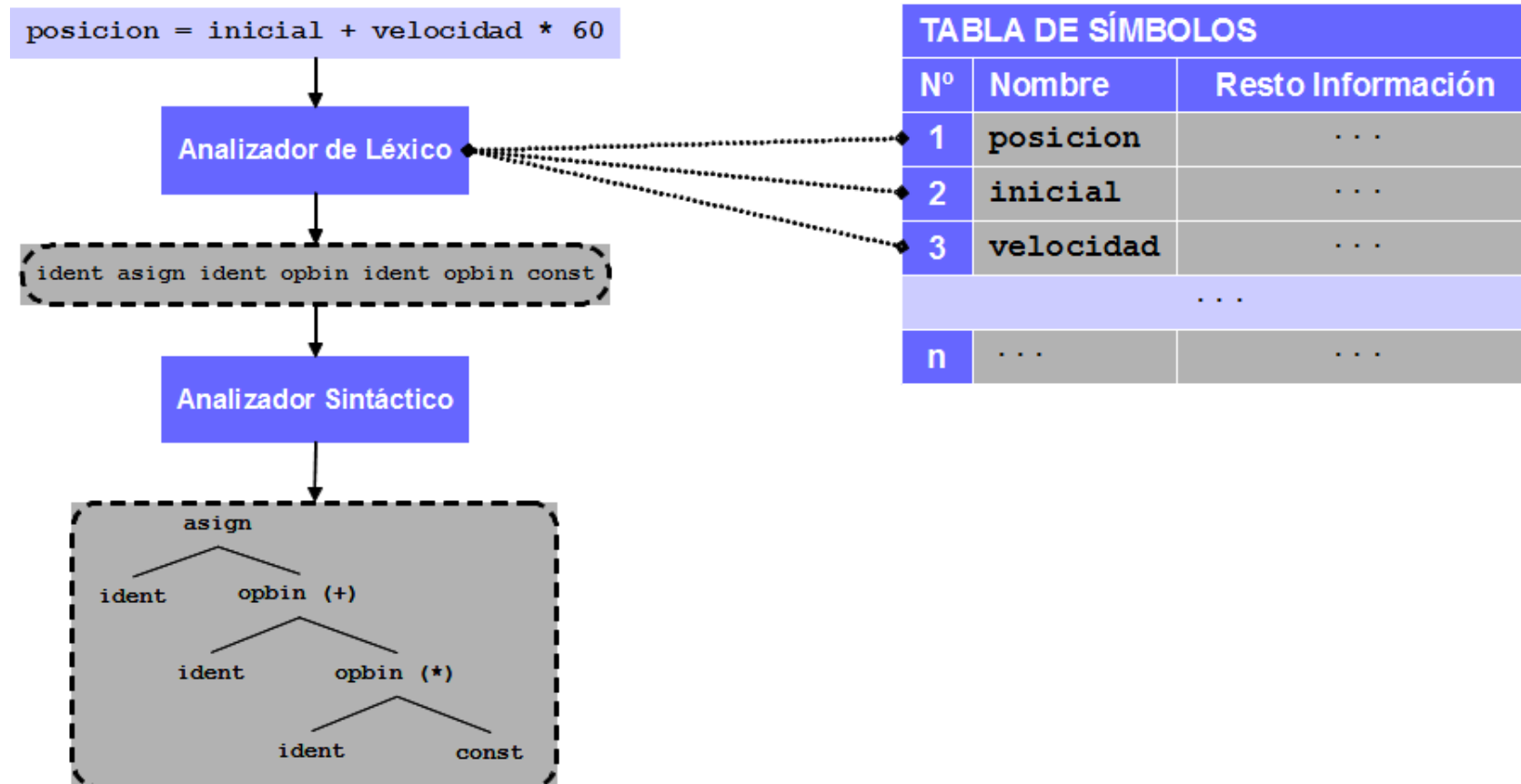
```
....  
for (i=0; i<1000; i++)  
{  
    r= 37.0-i*35;  
    b= 7.5;  
    z= b-sin(-r/35000);  
}
```

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

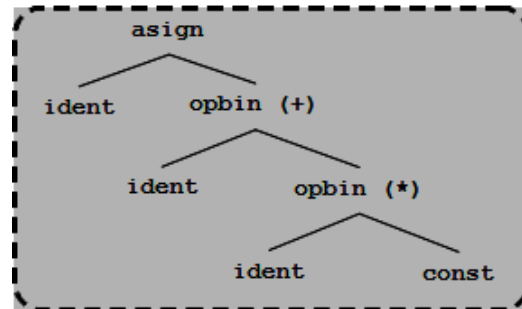


3.3 Fases de Traducción

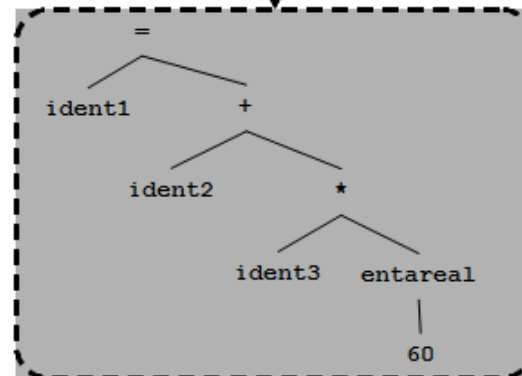
Fases de Traducción. Ejemplo (1/2) [Aho08]



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
```

Intérpretes

Intérprete: hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador.

Consecuencias inmediatas:

- No se crea un archivo o programa objeto almacenable en memoria para posteriores ejecuciones, es decir, cada vez que se ejecute el programa hay que volver a analizarlo.
- La ejecución del programa escrito en lenguaje fuente está supervisada por el intérprete.
- Las instrucciones de los bucles se analizan en cada iteración.
- La optimización solo se puede hacer a nivel de instrucción, no de estructuras, ni bloques, ni programas.
- **Ejemplo:** Bash.

Intérpretes

¿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.

Modelo de Memoria de un Proceso [Carr07] (pp.219-231)

Elementos responsables de la gestión de memoria:

- Lenguaje de programación.
- Compilador.
- Enlazador.
- Sistema operativo.
- Hardware para la gestión de memoria: MMU – Memory Management Unit.

Niveles de la Gestión de Memoria

Aumenta
el nivel
de
detalle



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

3.5 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 para 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 (por ejemplo, *plug-in*).

Modelo de Memoria de un Proceso [Carr07] (pp.246-251)

Estudiaremos aspectos relacionados con la gestión del mapa de memoria de un proceso, desde la generación del ejecutable a su carga en memoria:

- Nivel de regiones.
- Nivel de zonas.

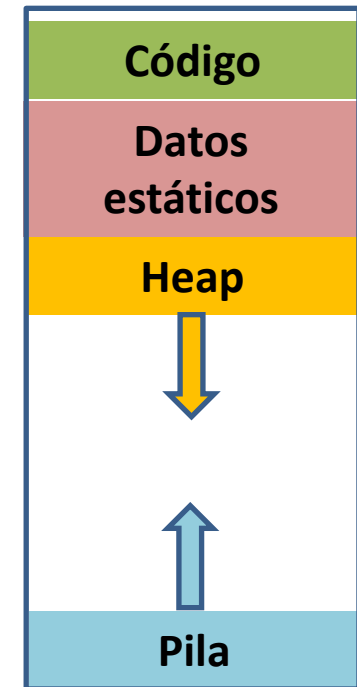
Para ello veremos:

- Implementación de tipos de objetos necesarios para un programa y su correspondencia con el mapa de memoria.
- Ciclo de vida de un programa.
- Estructura de un ejecutable.
- Bibliotecas.

Tipos de Datos (desde el punto de vista de su implementación en memoria)

- Datos estáticos:
 - Globales a todo el programa, módulo o locales a una función (ámbito de visibilidad de una variable).
 - Constantes o variables.
 - Con o sin valor inicial – implementación con direccionamiento absoluto o direccionamiento relativo (PIC – código independiente de la posición).
- Datos dinámicos asociados a la ejecución de una función:
 - Se almacenan en pila en un **registro de activación (contiene variables locales, parámetros, dirección de retorno)**.
 - Se crean al activar una función y se destruyen al terminar la misma.
- Datos dinámicos controlados por el programa – **heap** (zona de memoria usada tiempo de ejecución para albergar los datos no conocidos en tiempo de compilación).

Espacio de direcciones de un proceso



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

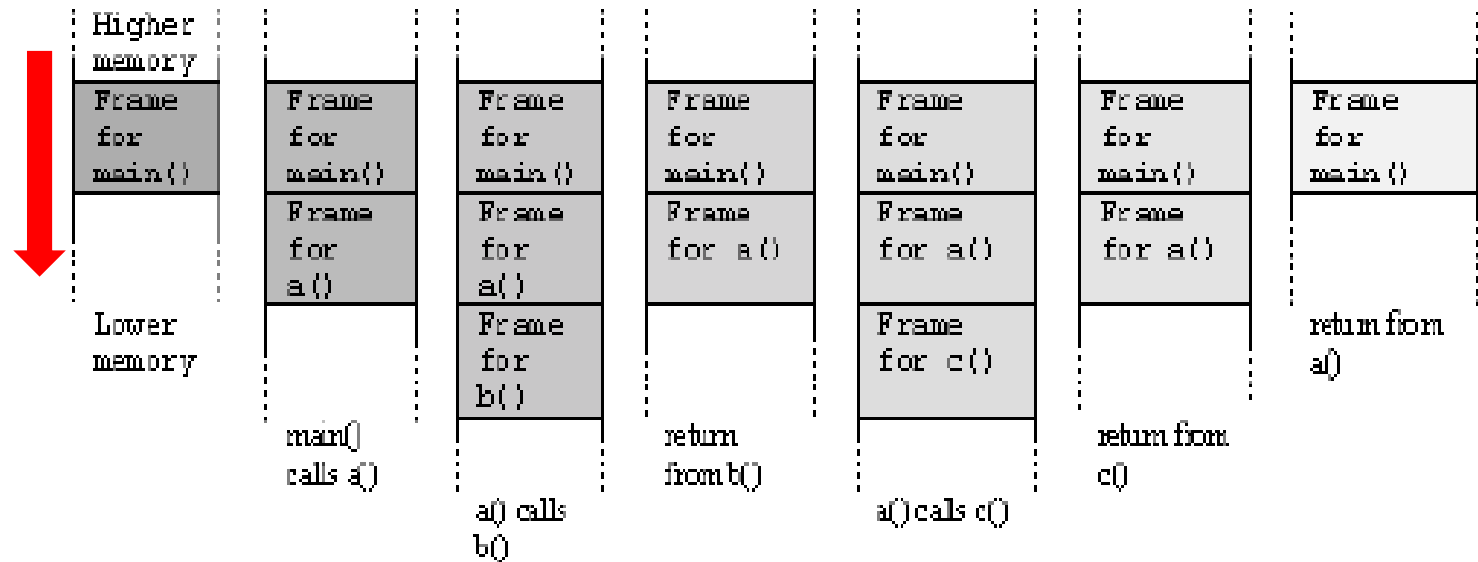
```
#include <stdio.h>
int a();
int b();
int c();
```

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

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

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

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



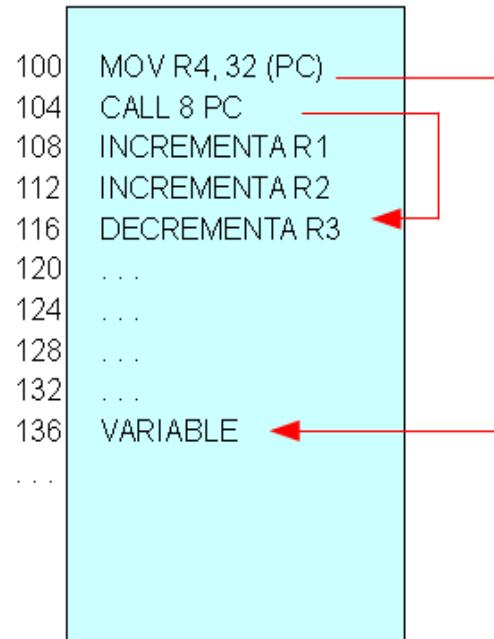
Frame: registro de activación o marco de pila.

Nota: en prácticas se verán las operaciones *down* y *up* para cambiar de marco en la depuración con *gdb*.

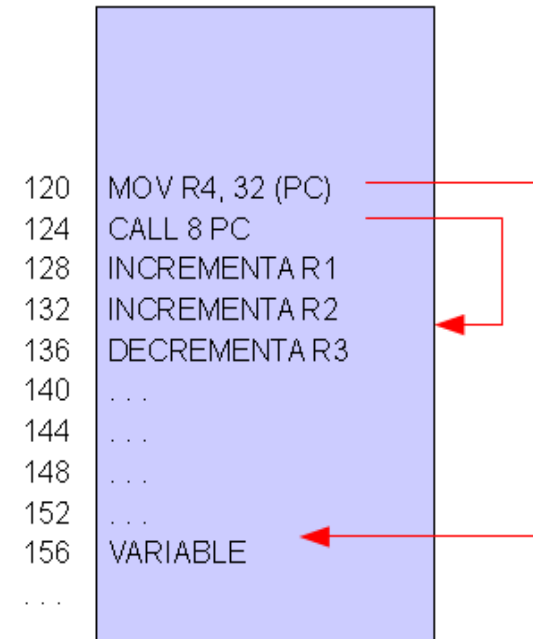
Código Independiente de la Posición (PIC, Position Independent Code)

- 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.

Memoria



Memoria



Suponemos instrucciones que ocupan **4 direcciones de memoria**, por lo tanto, el incremento del PC se produce en 4.

Ejemplos de diferentes Objetos de Memoria

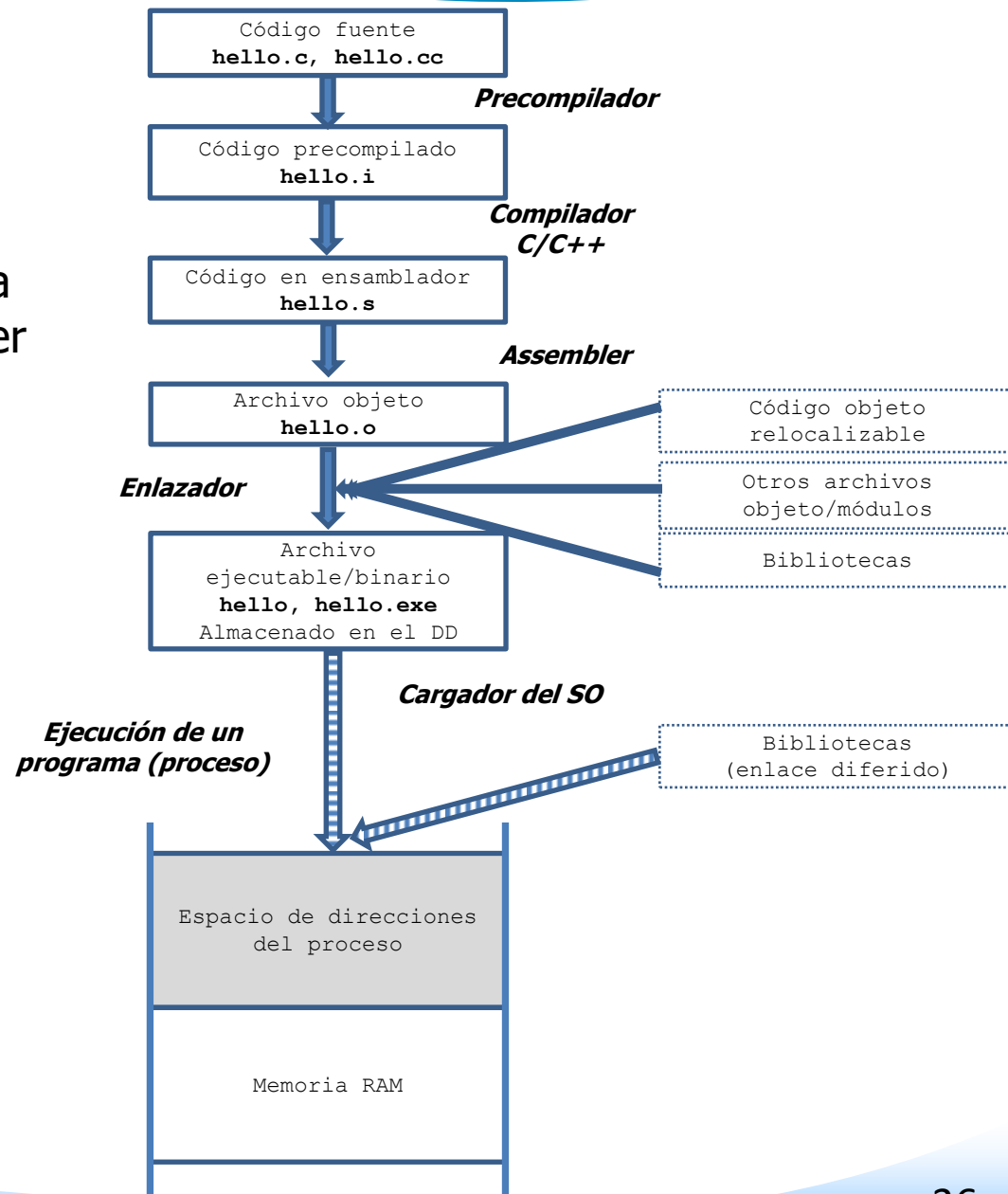
```
int a;           /* variable estática global sin valor inicial */
int b= 8;        /* variable estática global con valor inicial */
static int c;    /* variable estática de módulo sin valor inicial */
static int d= 8; /* variable estática de módulo con valor inicial */
const int e= 8;  /* constante estática global */
static const int f= 8; /* constante estática de módulo */
extern int g;    /* referencia a variable global de otro módulo */

void funcion (int h) /* parámetro: variable dinámica de función */
{
    int i;           /* variable dinámica de función sin valor inicial */
    int j= 8;        /* variable dinámica de función con valor inicial */
    static int k;     /* variable estática local sin valor inicial */
    static int l= 8;  /* variable estática local con valor inicial */
    {
        int m;       /* variable dinámica de bloque sin valor inicial */
        int n= 8;    /* variable dinámica de bloque con valor inicial */
    }
    . . .
}
```

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 (archivos .i en C).
2. Compilación (archivos .s en C).
3. Ensamblado (archivos .o en C).
4. Enlazado (archivos .exe y a.out).
5. Carga y Ejecución.



Ejemplo de Compilación

gcc/g++ es un wrapper (envoltorio) que invoca a:

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

Podemos salvar los archivos temporales con:

```
bash:~$ gcc -save-temps
```

Podemos generar el archivo ensamblador con:

```
bash:~$ gcc -S
```

El archivo objeto con:

```
bash:~$ gcc -c
```

Enlazar un objeto para generar el ejecutable con:

```
bash:~$ ld objeto.o -o eje
```

```
.section      __TEXT,__text,regular,pure_instructions
.macrosx_version_min 10, 13
.globl _main
.p2align      4, 0x90

_main:
.cfi_startproc
## BB#0:
pushq   %rbp
Lcfi0:
.cfi_def_cfa_offset 16
Lcfi1:
.cfi_offset %rbp, -16
movq     %rsp, %rbp
Lcfi2:
.cfi_def_cfa_register %rbp
subq     $16, %rsp
leaq     L_.str(%rip), %rdi
movl     _x(%rip), %esi
movb     $0, %al
callq    _printf
xorl     %esi, %esi
movl     %eax, -4(%rbp)      ## 4-byte Spill
movl     %esi, %eax
addq     $16, %rsp
popq     %rbp
retq
.cfi_endproc

.section      __DATA,__data
.globl _x
.p2align      2
_x:
.long     42                ## 0x2a

L_.str:
.section      __TEXT,__cstring,cstring_literals
.asciz  "Hola mundo, x = %d\n"

.subsections_via_symbols
```

Compilación

El compilador procesa cada uno de los archivos de código fuente para generar el correspondiente archivo objeto.

Realiza las siguientes acciones:

- 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** (necesita reubicación).
- Las referencias a símbolos dinámicos se resuelven usando direccionamiento relativo a pila para datos relacionados a la invocación de una función, o con direccionamiento indirecto para el heap. No necesitan reubicación al no aparecer en el archivo objeto.
- Genera la **Tabla de símbolos e información de depuración**.

3.6 Ciclo de Vida de un Programa

Ejemplo

Programa ejemplo:

```
#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
0000000000000000 T _main
                                U _printf
000000000000002c D _x
```

nm: orden Linux para ver las diferentes secciones de un archivo .o (y con opción `-a` se pueden ver símbolos que pueden depurarse):

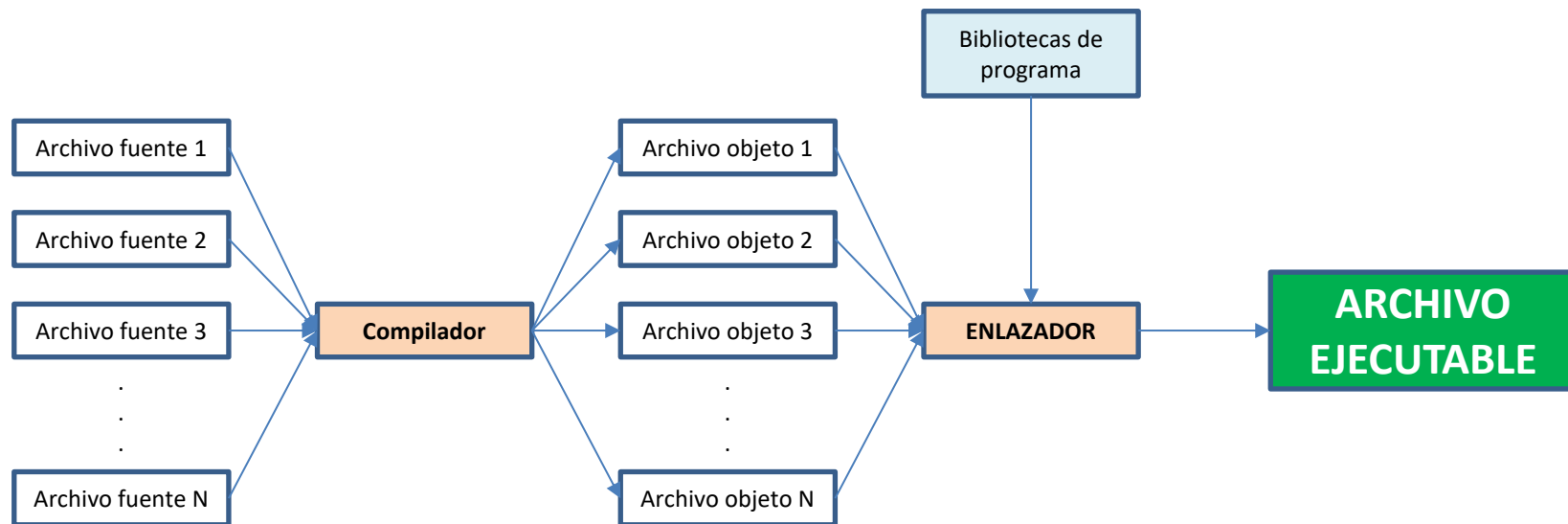
- T** indica sección de Texto (instrucciones máquina del programa).
- U** indica objetos no definidos en el programa (cadena de printf).
- D** indica sección de Datos (la variable global x).

3.6 Ciclo de Vida de un Programa

Enlazado

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

En ocasiones debe realizar reubicaciones dependiendo del esquema de gestión de memoria utilizado.

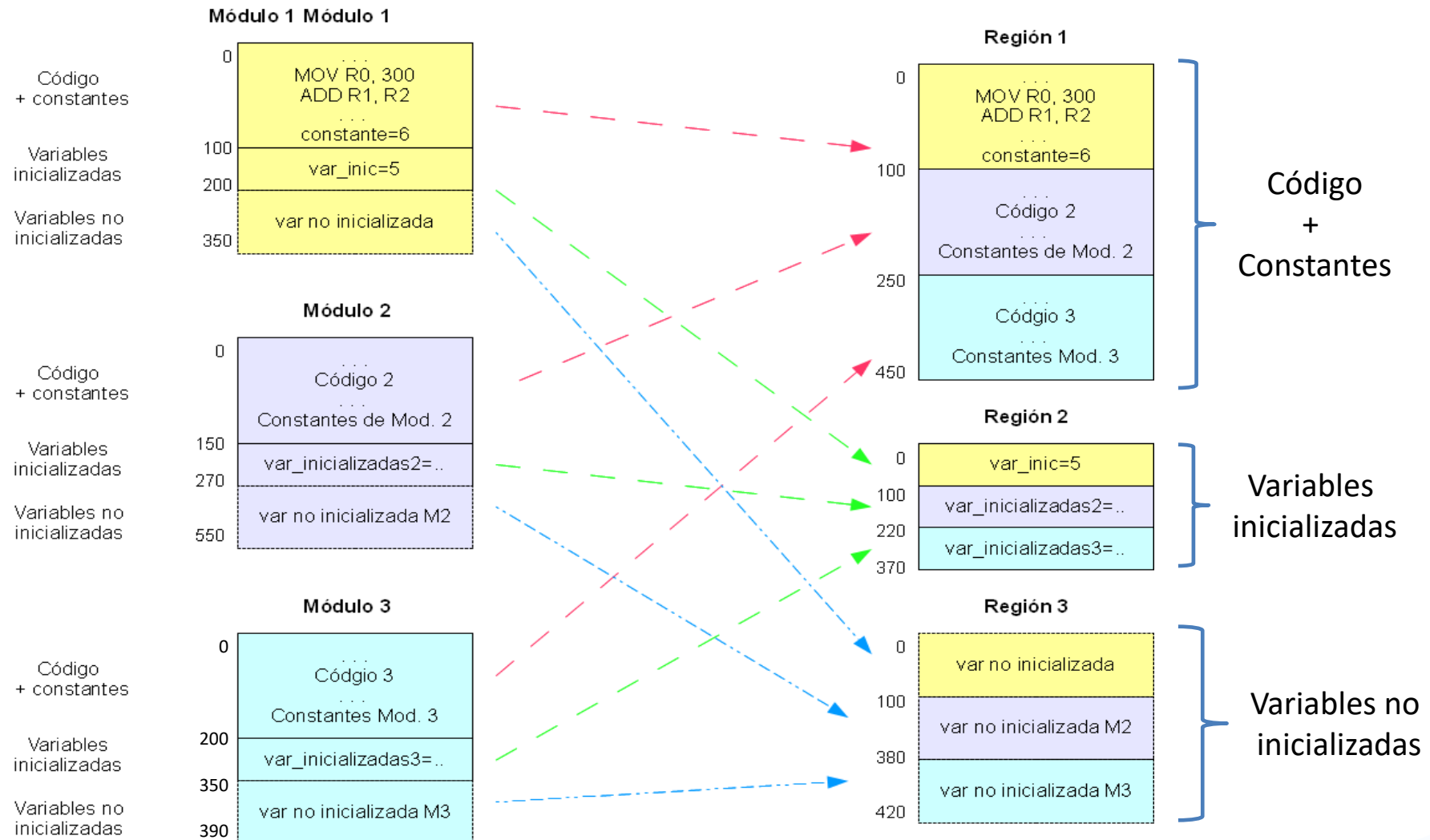


3.6 Ciclo de Vida de un Programa

Funciones del Enlazador

- Se completa la etapa de resolución de símbolos externos utilizando la tabla de símbolos.
- Se agrupan las zonas de características similares de los diferentes módulos en **regiones (código, datos inicializados o no, etc.)**.
- Se realiza **la reubicación de módulos** formando regiones – hay que transformar las referencias dentro de un módulo a referencias dentro de las regiones. Tras esta fase cada archivo objeto tiene una lista de reubicación que contiene los nombres de los símbolos y los desplazamientos dentro del archivo que deben aún parchearse.

Agrupamiento de módulos en regiones



Tipos de enlazado y ámbito

- **Atributos de enlazado:** externo, interno o sin enlazado.
- Los tipos de enlazado definen una especie de **ámbito**:
 - Enlazado externo --> visibilidad global.
 - Enlazado interno --> visibilidad de fichero.
 - Sin enlazado --> visibilidad de bloque.
- El tipo de enlazado indica si el mismo nombre (de una variable o función) en otro ámbito se refiere al mismo objeto o a otro distinto. Esto permite definir la visibilidad de los identificadores.

3.6 Ciclo de Vida de un Programa

Ejemplo

```
int x;
```

```
static st = 0;
```

```
void func(int);
```

```
int main()
```

```
{  
    for (x = 0; x < 10; x++)  
        func(x);  
}
```

```
void func(int j) {  
    st += j;  
    cout << st << endl;  
}
```

| Objeto | Tipo |
|--------|------------------|
| x | Enlazado externo |
| st | Enlazado interno |
| func | Enlazado externo |
| j | Sin enlazado |

Carga en memoria principal y Ejecución

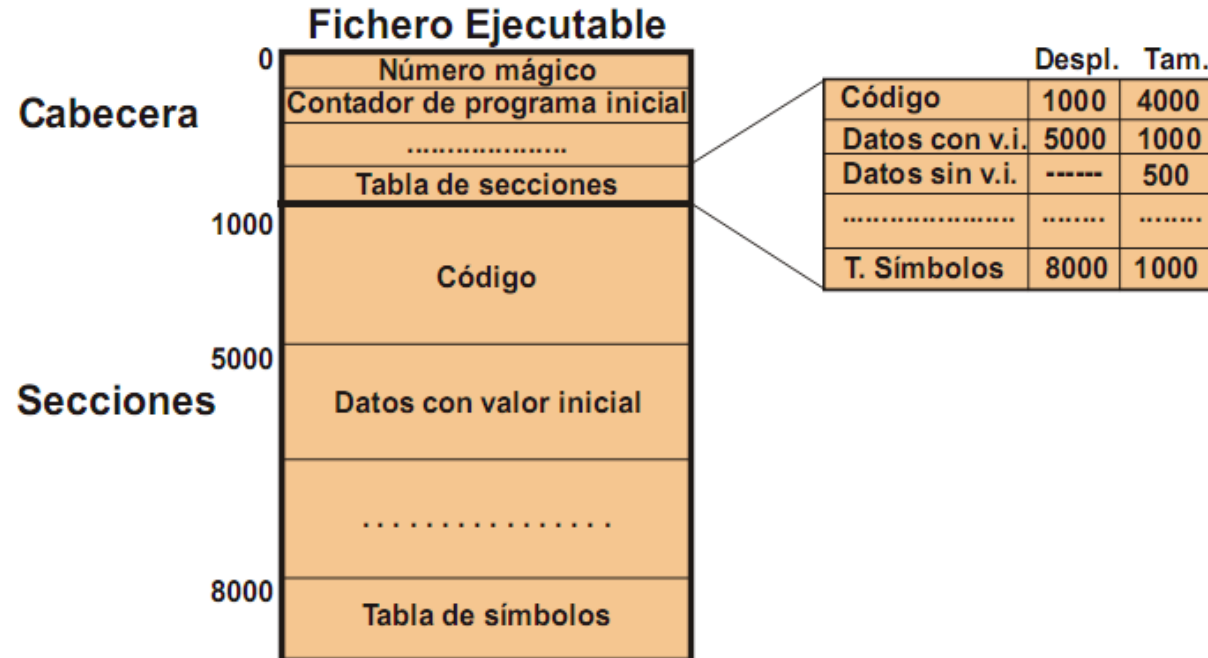
La reubicación del proceso se realiza en la **carga** (reubicación estática) o en **ejecución** (reubicación dinámica) y es función del Sistema Operativo ayudado por un hardware específico (MMU – Unidad de Gestión de Memoria). Depende del tipo de gestión de memoria que se realice:

- Paginación.
- Segmentación.

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.

Formato de archivo ejecutable



Formatos de archivo objeto y ejecutables

| | Descripción |
|-------|---|
| a.out | Es el formato original de los sistemas Unix. Consta de tres secciones: <code>text</code> , <code>data</code> y <code>bss</code> que se corresponden con el código, datos inicializados y sin inicializar. No tiene información para depuración. |
| COFF | El <i>Common Object File Format</i> posee múltiples secciones cada una con su cabecera pero están limitadas en número. Aunque permite información de depuración, ésta es limitada. Es el formato utilizado por Windows. |
| ELF | <i>Executable and Linking Format</i> es similar al COFF pero elimina algunas de sus restricciones. Se utiliza en los sistemas Unix modernos, incluido GNU/Linux y Solaris. |

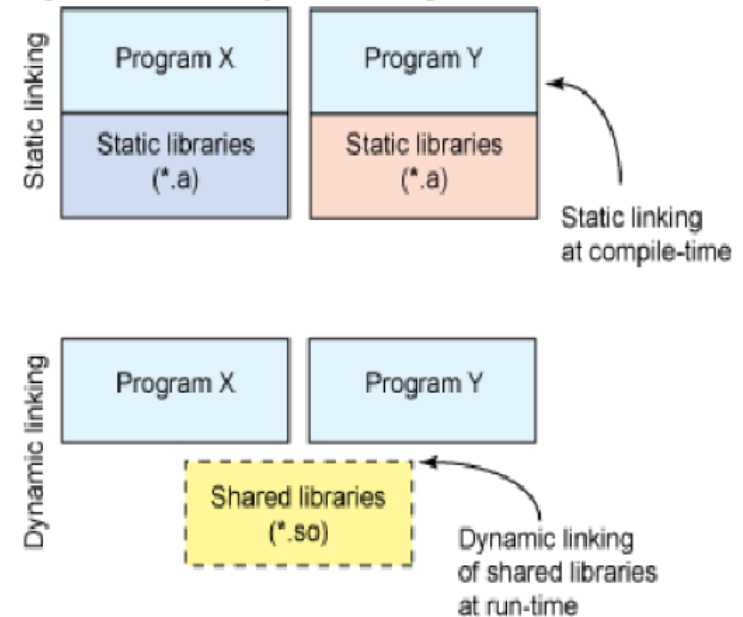
Ejemplos de secciones de un archivo ejecutable ELF (readelf)

- **.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** o **.rodata** – **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.

Definiciones [Carr07] (pp.262-264)

- **Biblioteca:** colección de objetos, normalmente relacionados entre sí.
- Las bibliotecas favorecen modularidad y reusabilidad de código.
- Podemos clasificarlas según la forma de enlazarlas:
 - **Bibliotecas estáticas** - se ligan con el programa en el enlazado (**.a**)
 - **Bibliotecas dinámicas** – se ligan con el programa en ejecución (**.so**)

Figure 2. Static vs. dynamic linking



Bibliotecas Estáticas

Una biblioteca estática es básicamente un conjunto de archivos objeto que se copian en un único archivo que forma la biblioteca.

Pasos para su creación:

- 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
```

Bibliotecas Estáticas

Inconvenientes de las bibliotecas estáticas:

- El código de la biblioteca está en todos los ejecutables que la usan, lo que desperdicia disco y memoria principal.
- Si actualizamos una biblioteca estática, debemos recompilar los programas que la usan para que se puedan beneficiar de la nueva versión.
- Producen ejecutables grandes.

Bibliotecas Dinámicas

Las bibliotecas dinámicas se integran con los procesos que las usan en tiempo de ejecución, por ello se realiza previamente la reubicación de módulos.

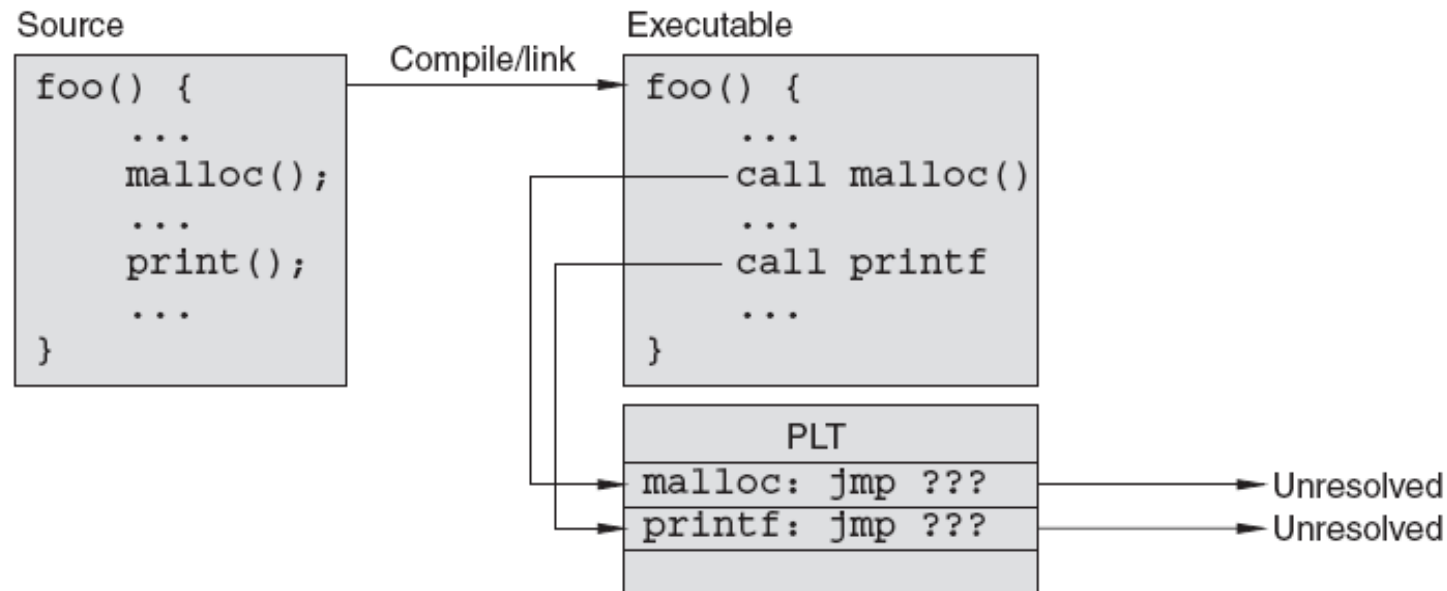
Resuelven los inconvenientes que presentan las bibliotecas estáticas.

El archivo correspondiente a una biblioteca dinámica se diferencia de un archivo ejecutable en los siguientes aspectos:

- Contiene información de reubicación.
- Contiene una tabla de símbolos.
- En la cabecera no se almacena información de punto de entrada.

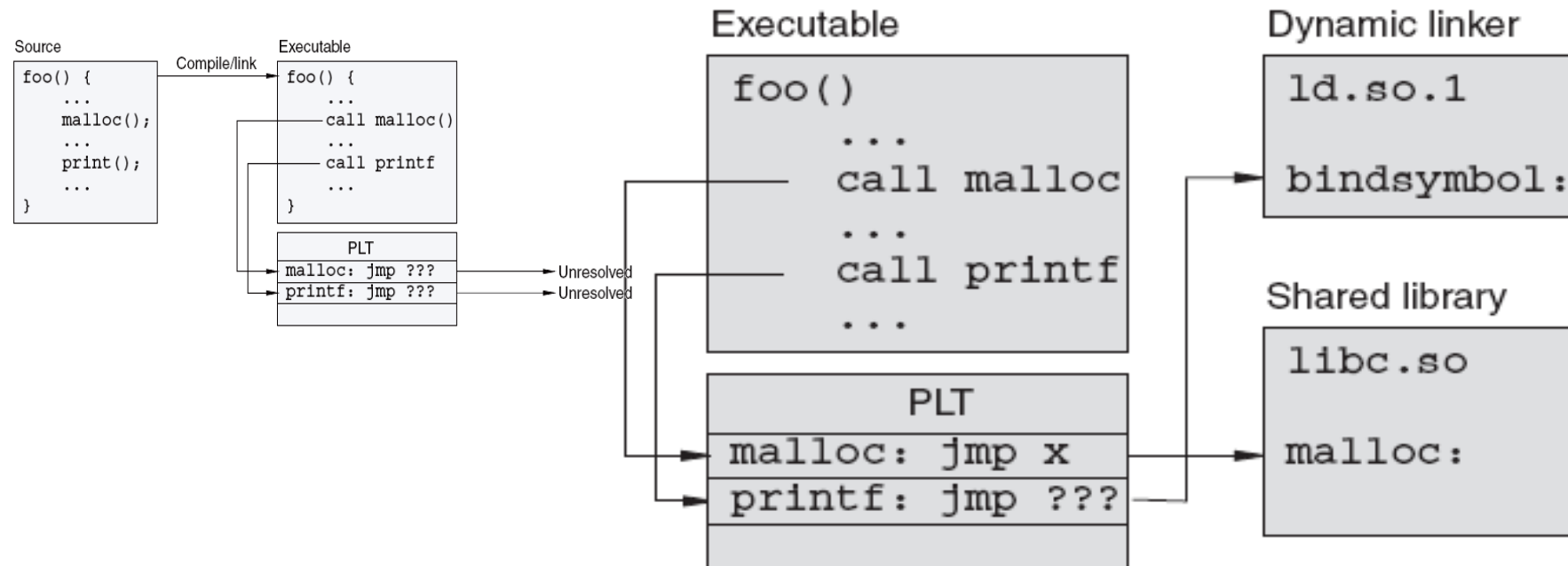
Al usar una biblioteca dinámica, en el proceso de montaje del programa ejecutable se incluye **un módulo de montaje dinámico (enlazador dinámico)**: carga y monta las bibliotecas dinámicas usadas por el programa durante su ejecución.

Estructura de un ejecutable que usa bibliotecas dinámicas (1/2)



PTL – Procedure Linkage Table en ELF (tabla de símbolos e información de reubicación)

Estructura de un ejecutable que usa bibliotecas dinámicas (2/2)



ld.so.1 es el enlazador dinámico que está en el código del ejecutable
x es una dirección física de memoria principal donde comienza el código de la función **malloc**

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
```

- La orden para ver las bibliotecas que están enlazadas con un programa es:

```
ldd hola
```


3.8 Automatización del Proceso de Compilación y Enlazado.

Automatizar la construcción es la técnica utilizada durante el ciclo de vida de desarrollo de software donde la transformación del código fuente en el ejecutable se realiza mediante un guión (script).

La automatización mejora la calidad del resultado final y permite el control de versiones.

Varias formas:

- Herramienta **make** – archivos makefile.
- IDE (Integrated Development Environment – Entornos de Desarrollo Integrados), que embebe los guiones y el proceso de compilación y enlazado, p.e. **CodeBlocks**.