

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.**

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. |

8-ene-2013

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.
- Una **sentencia** en un lenguaje de alto nivel da lugar, tras el proceso de traducción, a **varias instrucciones** en lenguaje máquina.
- Algo expresado en un lenguaje de alto nivel utiliza **notaciones más cercanas** a las **habituales** 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

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.

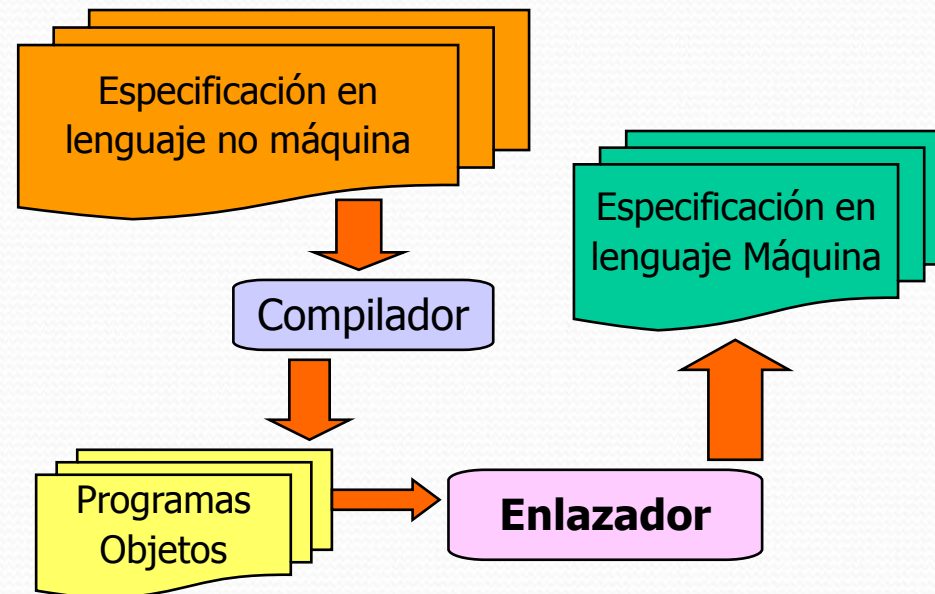
La forma en la que un programa escrito para una máquina virtual es posible ejecutarlo en una máquina real puede ser:

- Compilador.
- Intérprete.

Definición de Compilador

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

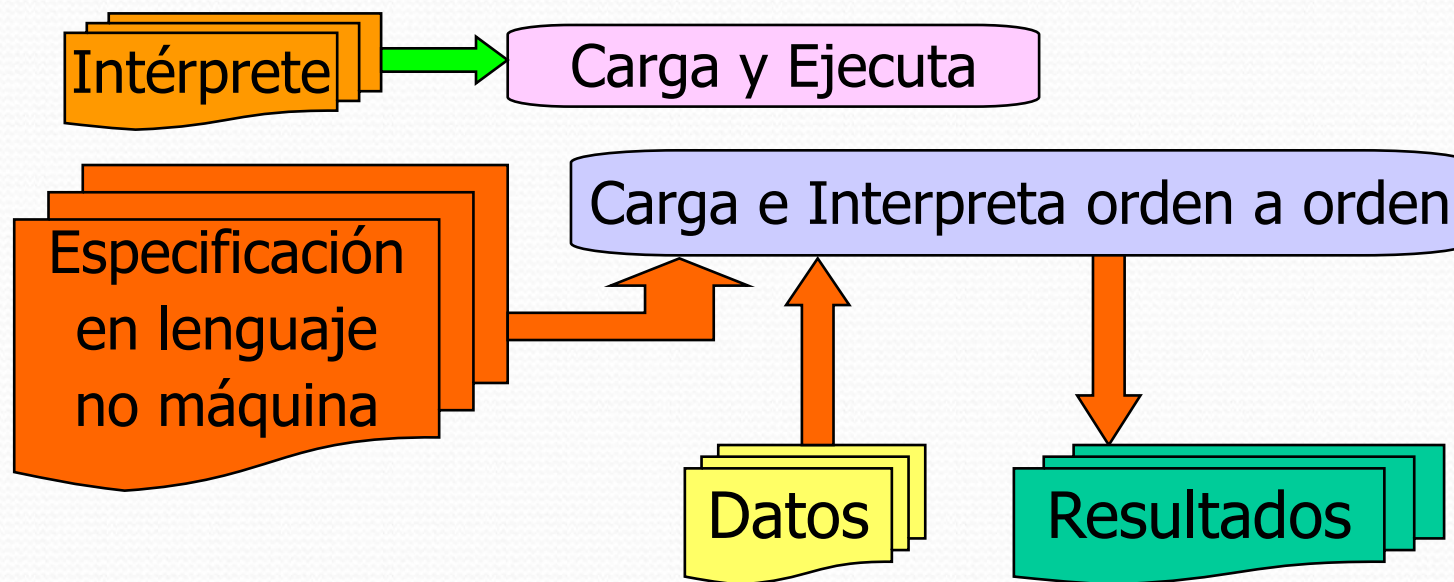
Enlazador (linker) realiza el enlazado de los programas completando las instrucciones máquina necesarias (añade rutinas binarias de funcionalidades no programadas directamente en el programa fuente) y generando un programa ejecutable para la máquina real.



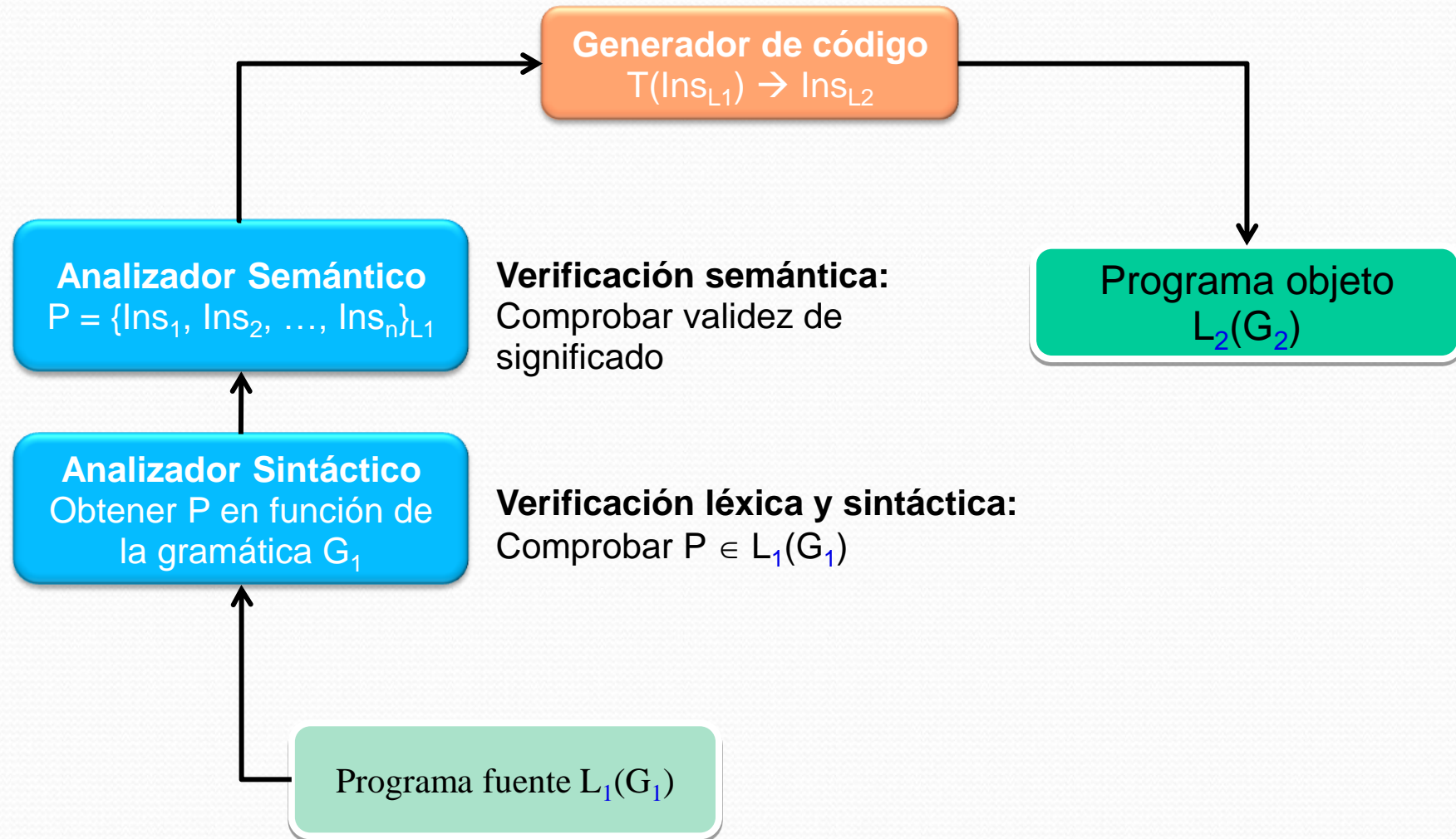
Definición de Intérprete

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



3.2 Construcción de Traductores

Definición de Gramática

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

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.

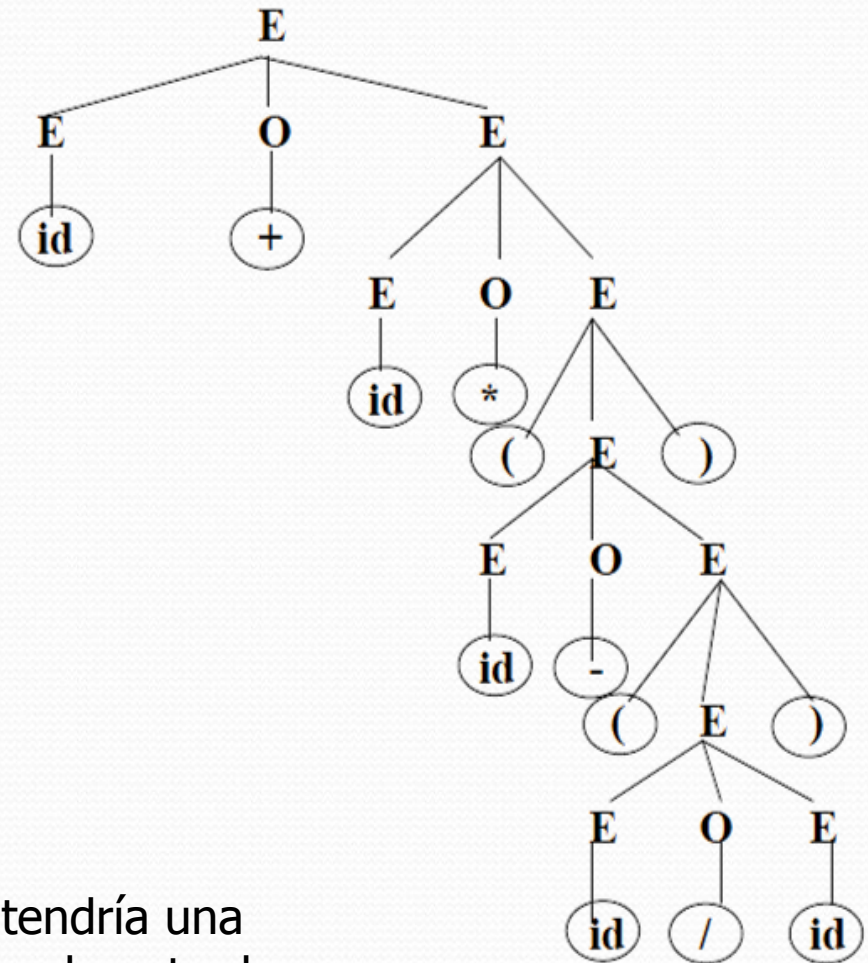
Ejemplo

Dada la gramática siguiente:

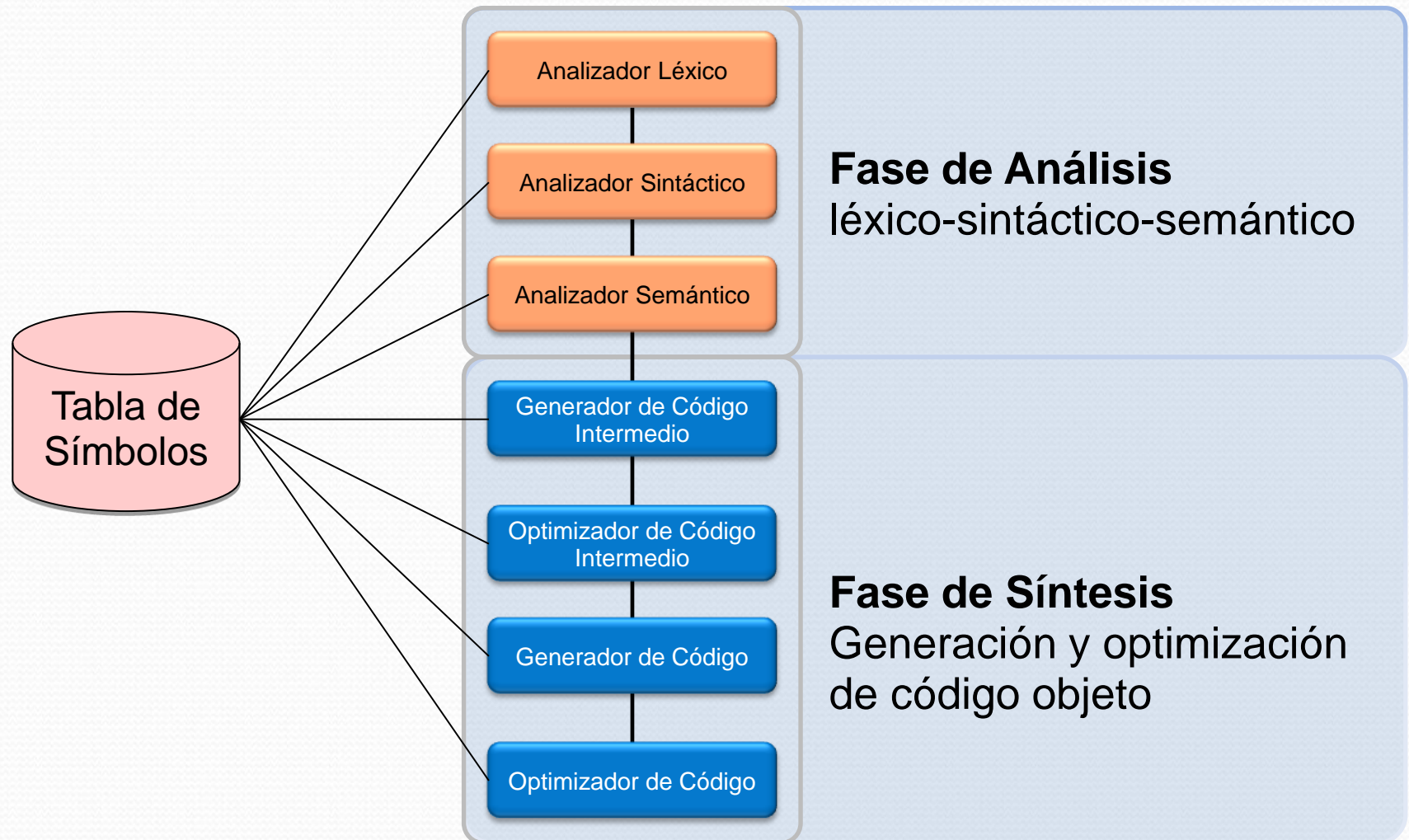
$$\begin{aligned} P &= \{ E \rightarrow E O E \\ &\quad \quad \quad | (E) \\ &\quad \quad \quad | id \\ O &\rightarrow + \mid - \mid * \mid / \\ \} \\ V_N &= \{E, O\} \\ V_T &= \{ (,), id, +, -, *, / \} \\ S &= E \end{aligned}$$

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.



Fases en la construcción de un traductor

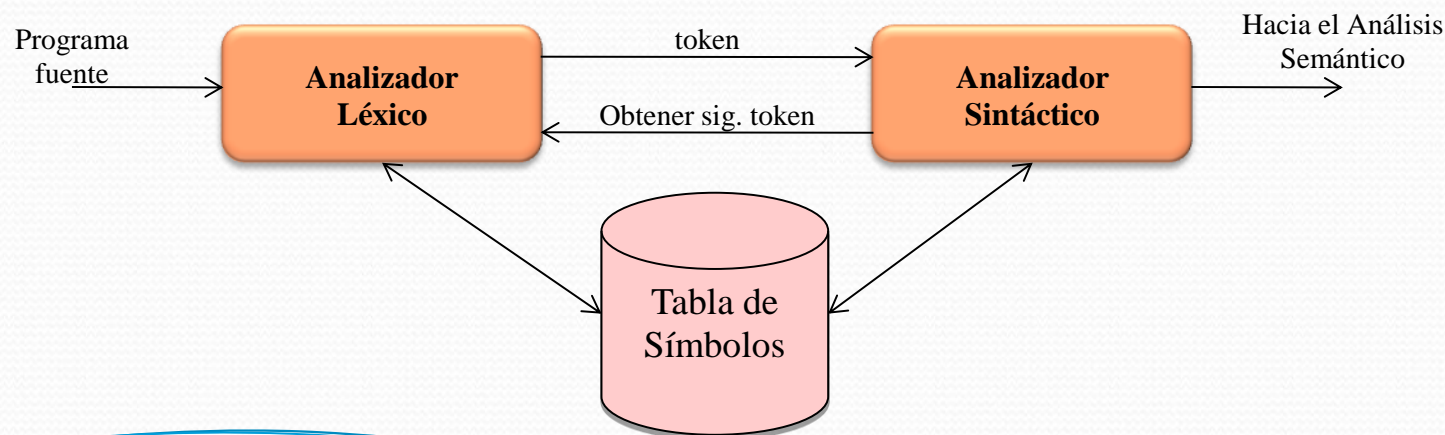


Análisis Léxico. Función Principal y Conceptos

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.

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.
- **Patrón**: Descripción de la forma que pueden tomar los lexemas de un token.



Análisis Léxico. Función Principal y Conceptos. Ejemplo 1

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>
IDENT	Letra seguida por letras y dígitos	<code>pi, dato1, dato3, D3</code>
NUMERO	Cualquier constante numérica	<code>0, 210, 23.45, 0.899, ...</code>

Análisis Léxico. Función Principal y Conceptos. Ejemplo 2

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

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 Léxico. Función Principal y Conceptos. Error Léxico.

En muchos lenguajes de programación, se cubren 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).
- **Tokens** para cada **signo de puntuación** (paréntesis, llaves, coma, punto, punto y coma, corchetes, ...).

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

3.3 Fases de Traducción

Análisis Léxico. Especificación de los Tokens

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. Destacamos:

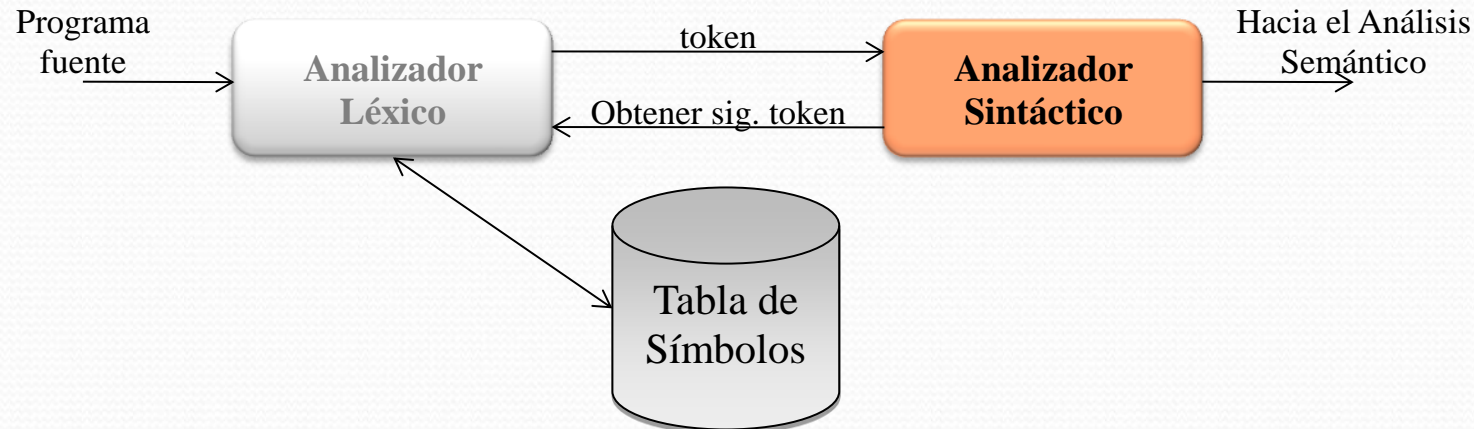
- Una gramática proporciona una especificación sintáctica precisa de un lenguaje de programación.
- A partir de ciertas clases gramaticales, es posible construir de manera automática un analizador sintáctico eficiente.
- Permite revelar ambigüedades sintácticas y puntos problemáticos en el diseño del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

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 $G = (V_N, V_T, P, S)$, 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.

se dice que es una *gramática libre de contexto* cuando el conjunto de producciones P obedece al formato:

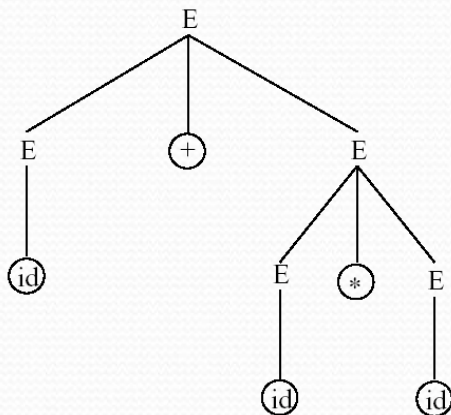
$$P = \{A \rightarrow \alpha / A \in V, \alpha \in (V_N \cup V_T)^*\}$$

es decir, sólo admiten tener un símbolo no terminal en su parte izquierda. La denominación *libre de contexto* se debe a que se puede cambiar A por α , independientemente del contexto en el que aparezca A .

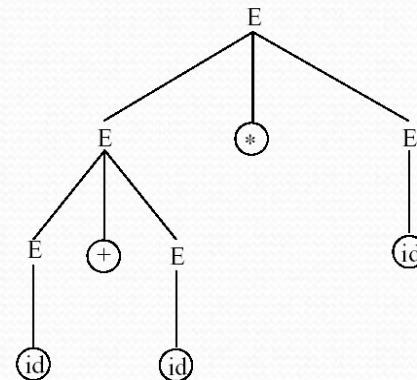
Análisis Sintáctico. Gramáticas ambiguas

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

Ejemplo 3.2: Dadas las producciones de la gramática del ejemplo 4.1 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

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 a una variable, el operador de asignación “=” y al delimitador de sentencia “;” respectivamente, deben cumplirse las siguiente reglas semánticas:

- **IDENTIFICADOR** debe estar previamente declarado.
- El tipo de la **expresion** 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** (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).

3.3 Fases de Traducción


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.

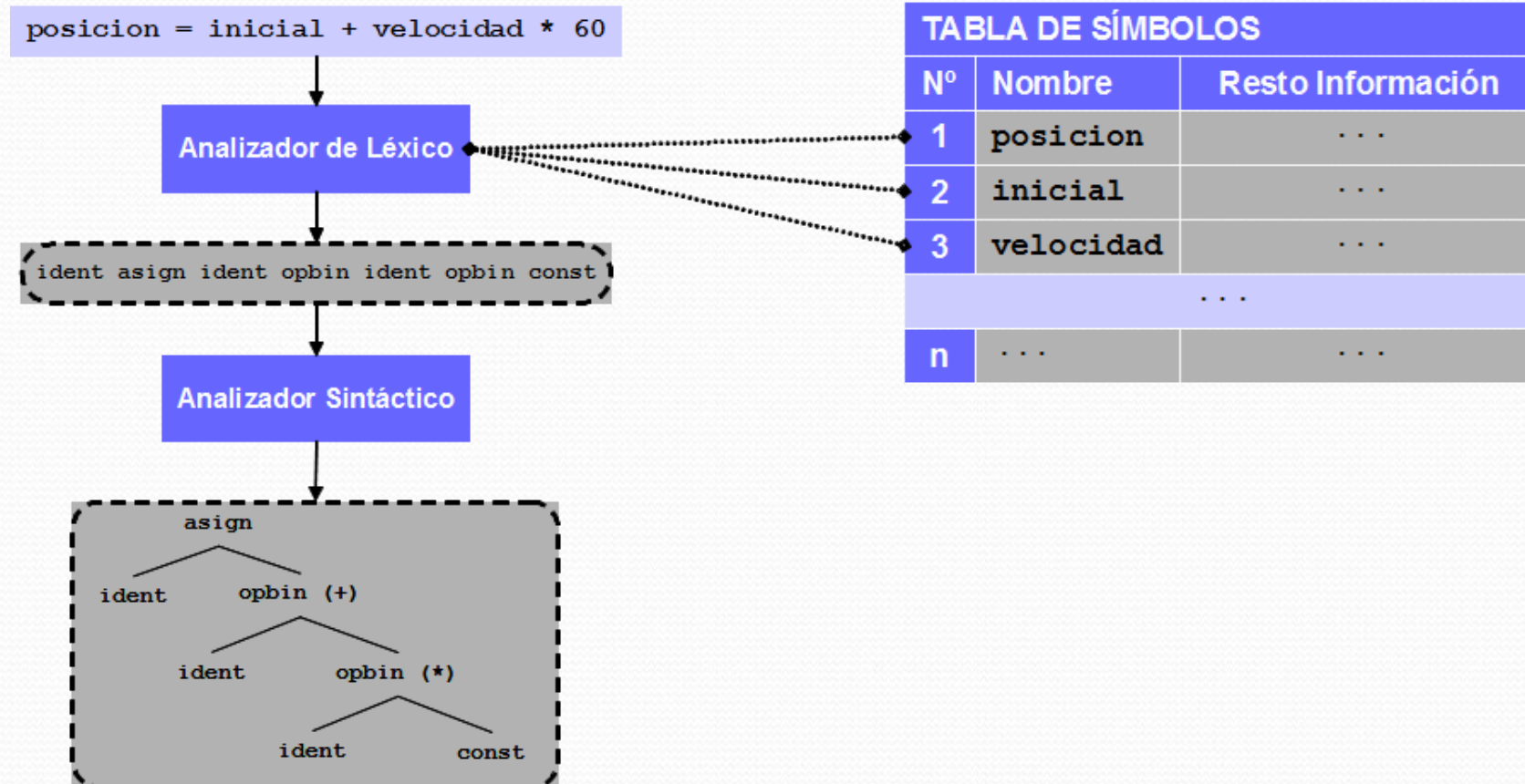
Ejemplo: Una asignación dentro de un bucle `for` en lenguaje C:

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

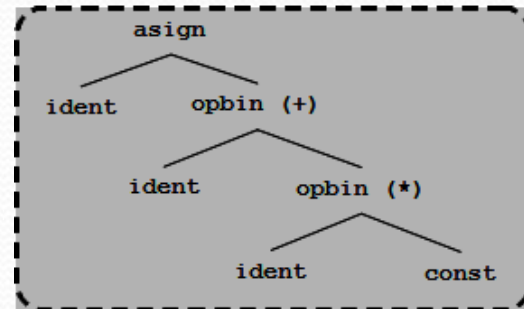
```
....  
b= 7.5;  
for (i=0; i<1000; i++)  
{  
    r= 37.0-i*35;  
    z= b-sin(-r/35000);  
}
```



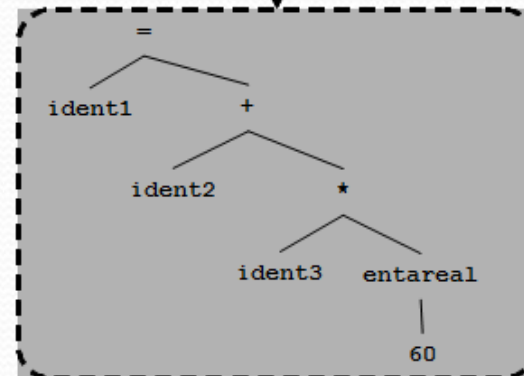
Fases de Traducción. Ejemplo (1/2) [Aho08]



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.
- La ejecución del programa escrito en lenguaje fuente está supervisada por el intérprete.
- 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
- MMU – Memory Management Unit

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.

3.5 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 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.

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 región.
- Nivel de zona.

Para ello veremos:

- Implementación de tipos de objetos necesarios por un programa.
- Ciclo de vida de un programa.
- Estructura de un ejecutable.
- Bibliotecas.

Tipos de Datos

- Datos estáticos:
 - Globales
 - Constantes o variables
 - Con o sin valor inicial – direccionamiento relativo: PIC
- 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)**
- Datos dinámicos controlados por el programa – heap

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

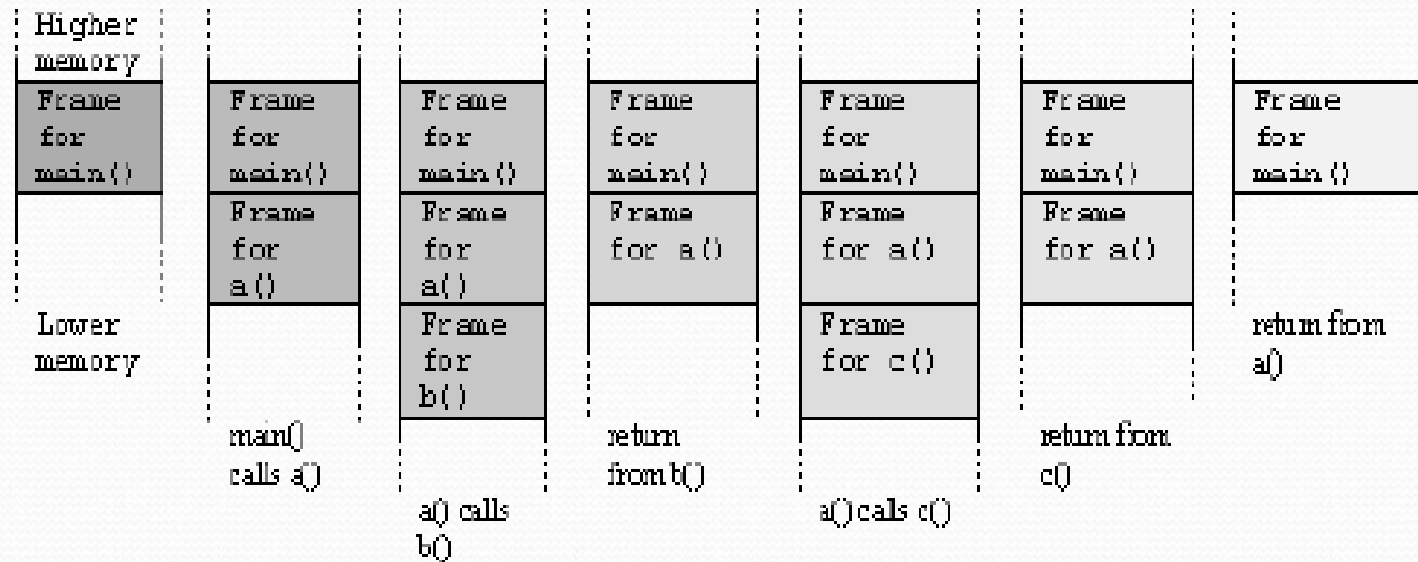
```
#include <stdio.h>
int a();
int b();
int c();
```

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

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

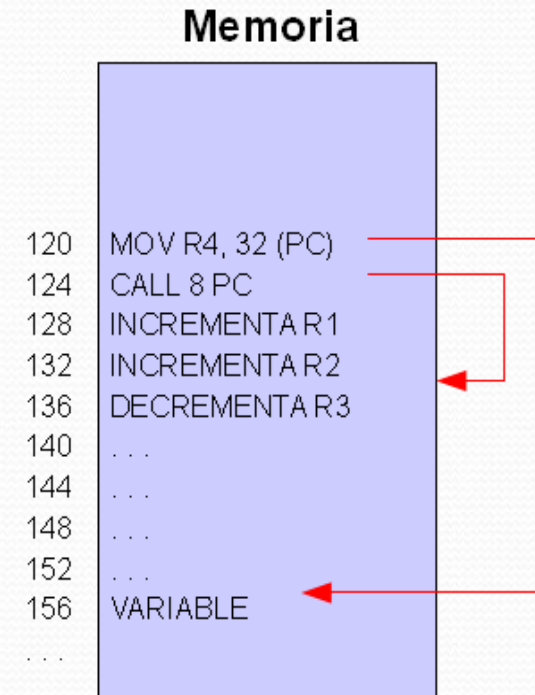
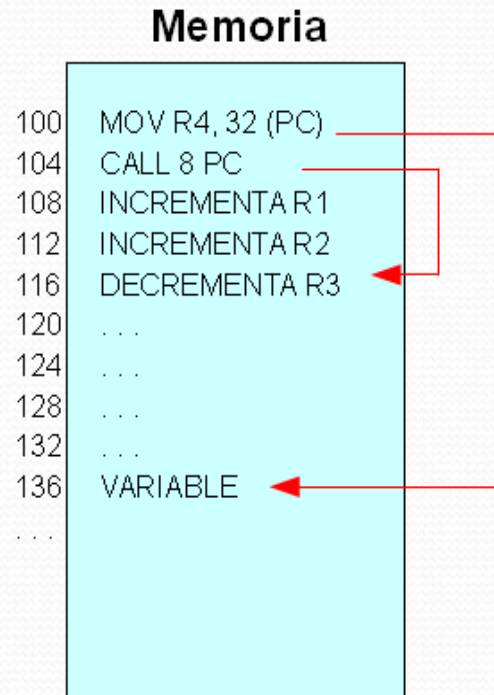
```
int c()
{ return 0; }
```

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



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.



Ejemplos de Tipos de 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 */
    }
    . . . .
}
```

Programa que usa los tres Tipos de Objetos de Memoria Básicos

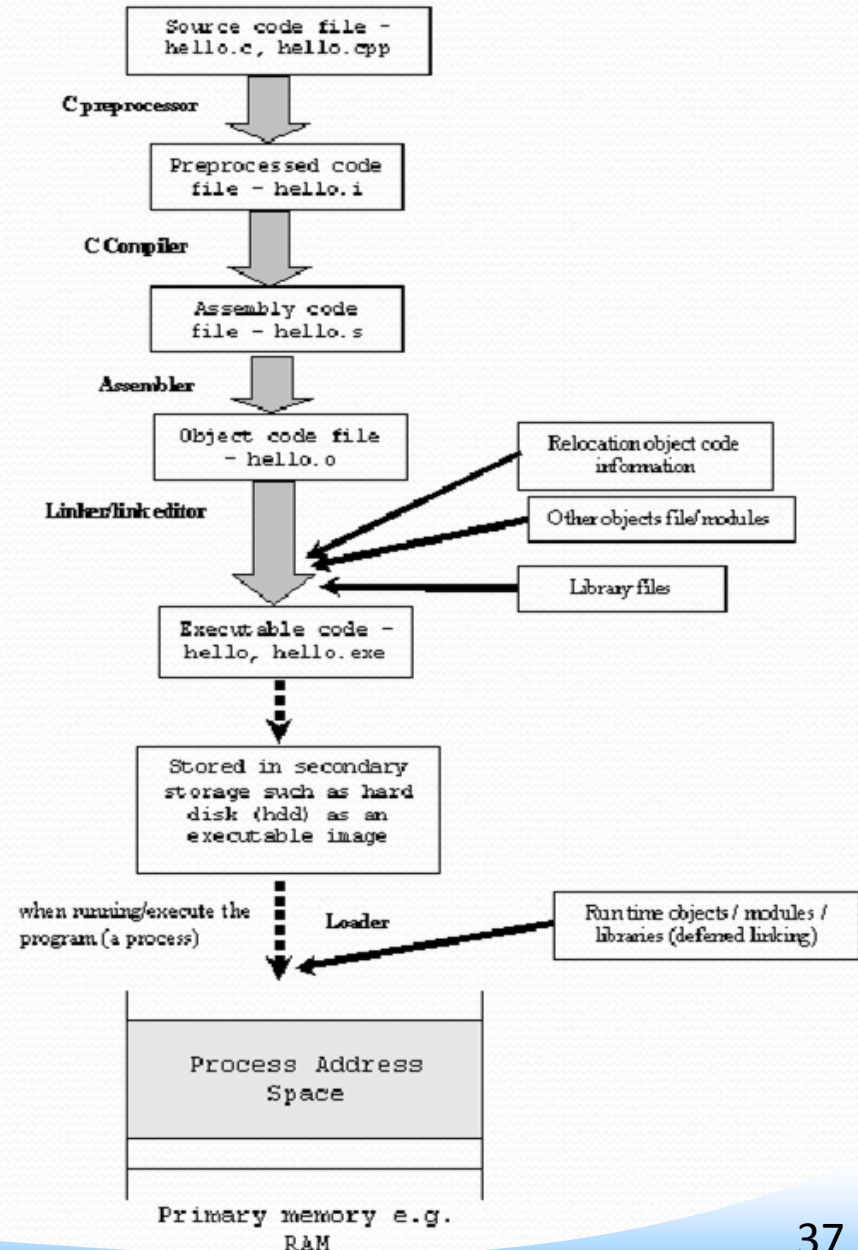
```
struct tipo {  
    int a, b;  
};  
  
int main (int argc, char *argv[])  
{  
    static struct tipo var_estatica;  
    struct tipo var_dinamica;  
    struct tipo *var_heap= malloc (sizeof (struct tipo));  
  
    var_estatica.b= 12;           /* 1 acceso con direccionamiento absoluto */  
    var_dinamica.b= 14;          /* 1 acceso con direccionamiento relativo a SP */  
    var_heap->b= 22;             /* 2 accesos con direccionamiento indirecto */  
  
    return 0;  
}
```


3.6 Ciclo de Vida de un Programa

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



3.6 Ciclo de Vida de un Programa

Ejemplo de Compilación

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

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

Podemos salvar los archivos temporales con

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

Podemos generar el archivo ensamblador con

```
$ gcc -S
```

El archivo objeto con

```
$ gcc -c
```

Enlazar un objeto para generar el ejecutable con:

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


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);
}
```

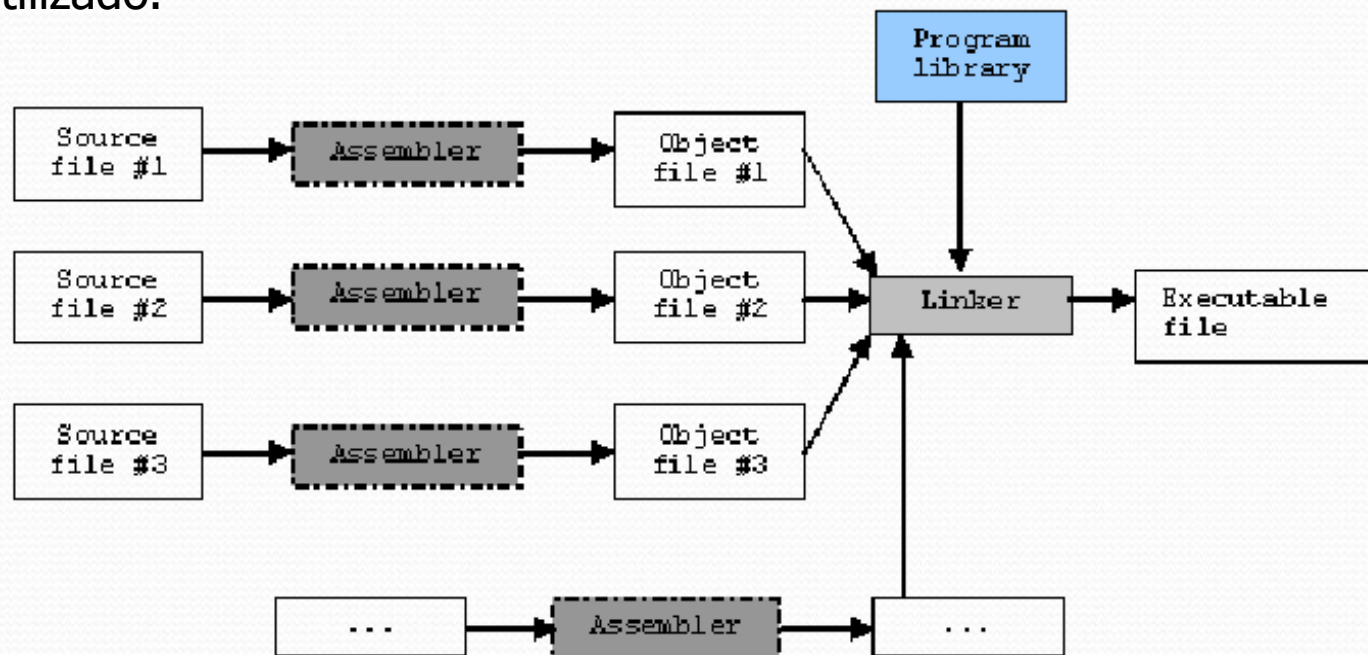
Tabla de símbolos:

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


Enlazado

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

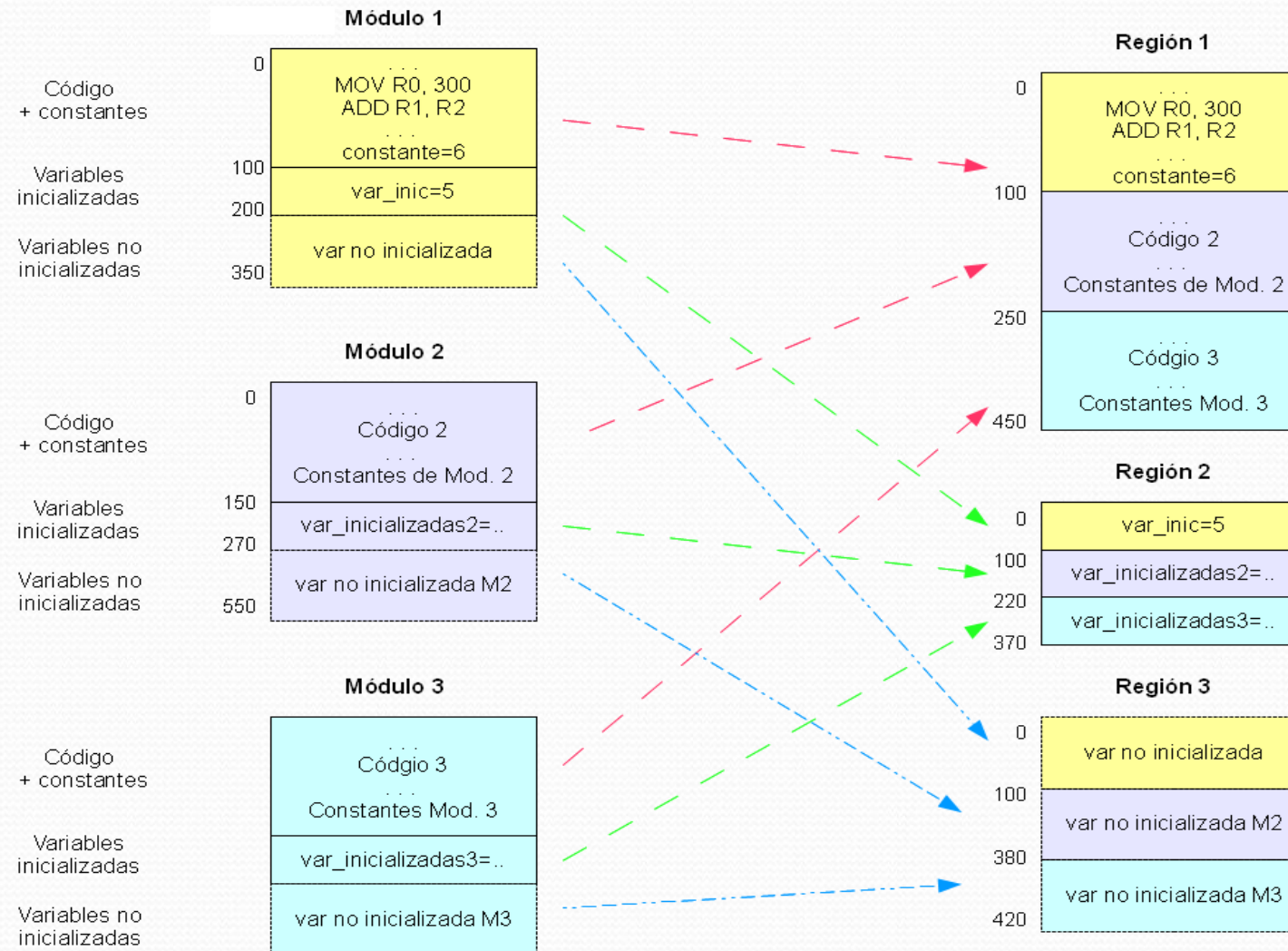
En ocasiones, debe realizar reubicaciones dependiendo del esquema de gestión de memoria utilizado.



Funciones del Enlazador

- Se completa la etapa de resolución de símbolos externos utilizando la tabla de símbolos.
- Se agrupan las regiones de similares características de los diferentes módulos en **regiones (código, datos inicializados o no, etc.)**
- Se realiza **la reubicación de módulos** – 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.
- En sistemas paginados, se realiza la **reubicación de regiones**, es decir, transformar direcciones de una región en direcciones del mapa del proceso.

Agrupamiento de módulos en regiones



3.6 Ciclo de Vida de un Programa

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

3.6 Ciclo de Vida de un Programa

Reglas de enlazado (1/3)

1. Cualquier objeto/identificador que tenga ámbito global deberá tener enlazado interno si su declaración contiene el especificador **static**.
2. Si el mismo identificador aparece con enlazados externo e interno, dentro del mismo fichero, tendrá enlazado externo.
3. Si en la declaración de un objeto o función aparece el especificador de tipo de almacenamiento **extern**, el identificador tiene el mismo enlazado que cualquier declaración visible del identificador con ámbito global. Si no existiera tal declaración visible, el identificador tiene enlazado externo.

Reglas de enlazado (2/3)

4. Si una función es declarada sin especificador de tipo de almacenamiento, su enlazado es el que correspondería si se hubiese utilizado **extern** (es decir, **extern** se supone por defecto en los prototipos de funciones).
5. Si un objeto (que no sea una función) de ámbito global a un fichero es declarado sin especificar un tipo de almacenamiento, dicho identificador tendrá enlazado externo (ámbito de todo el programa). Como excepción, los objetos declarados **const** que no hayan sido declarados explícitamente **extern** tienen enlazado interno.

3.6 Ciclo de Vida de un Programa

Reglas de enlazado (3/3)

6. Los identificadores que respondan a alguna de las condiciones que siguen tienen un atributo sin enlazado:
- Cualquier identificador distinto de un objeto o una función (por ejemplo, un identificador **typedef**).
 - Parámetros de funciones.
 - Identificadores para objetos de ámbito de bloque, entre corchetes **{}**, que sean declarados sin el especificador de clase **extern**.

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

3.6 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:

- El cargador **copia el programa en memoria sin modificarlo**. Es la **MMU** la encargada de realizar la **reubicación en ejecución**.
- En **paginación**, el **hardware** es capaz de **reubicar los procesos en ejecución** por lo que el cargador lo carga sin modificación.
- Si no usamos hardware de reubicación, ésta se realiza en la **carga**.

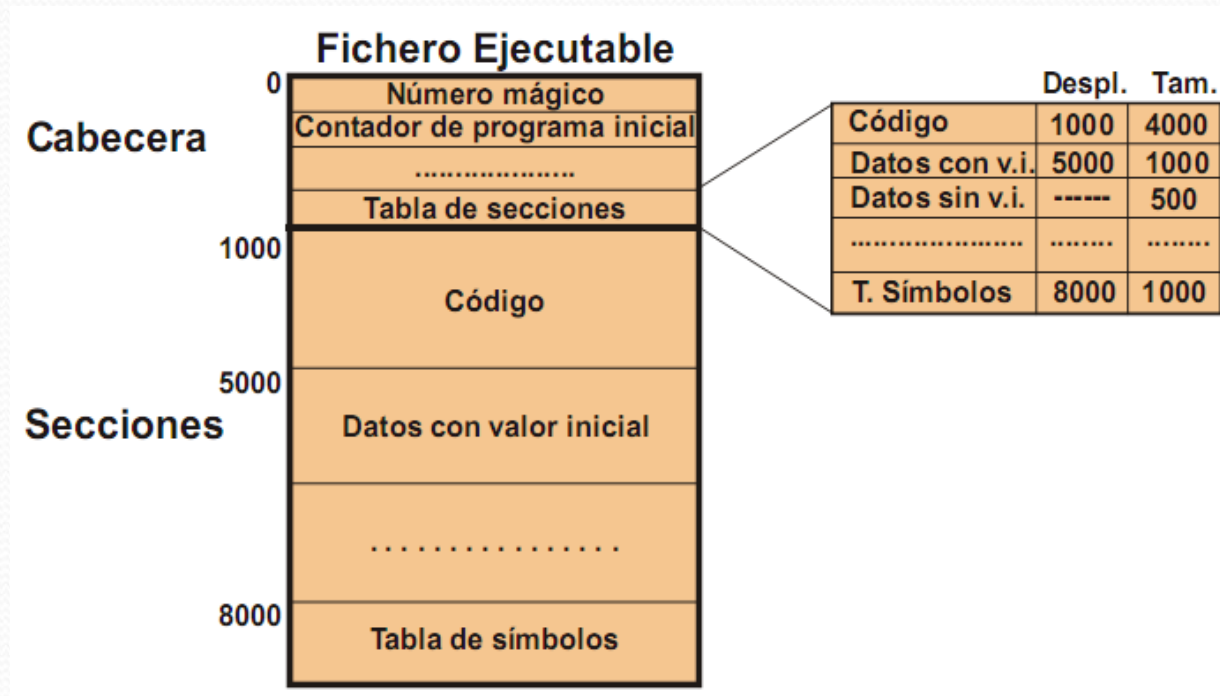
3.6 Ciclo de Vida de un Programa

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.

3.6 Ciclo de Vida de un Programa

Formato de archivo ejecutable



3.6 Ciclo de Vida de un Programa

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.

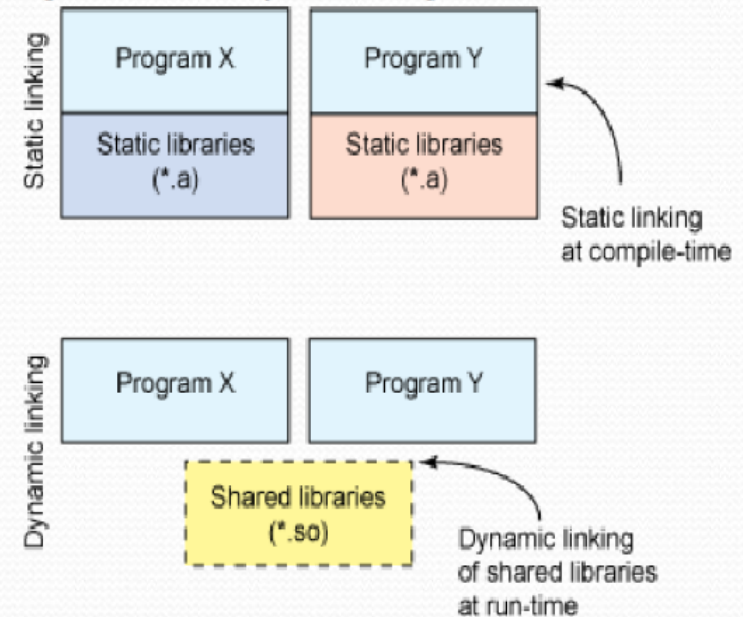
Secciones de un archivo

- `.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.

Definiciones [Carr07] (pp.262-267)

- **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 enlazan con el programa en la compilación (.a)
 - **Bibliotecas dinámicas** – se enlazan 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.

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 Dinámicas

Las bibliotecas estáticas tiene algunos inconvenientes:

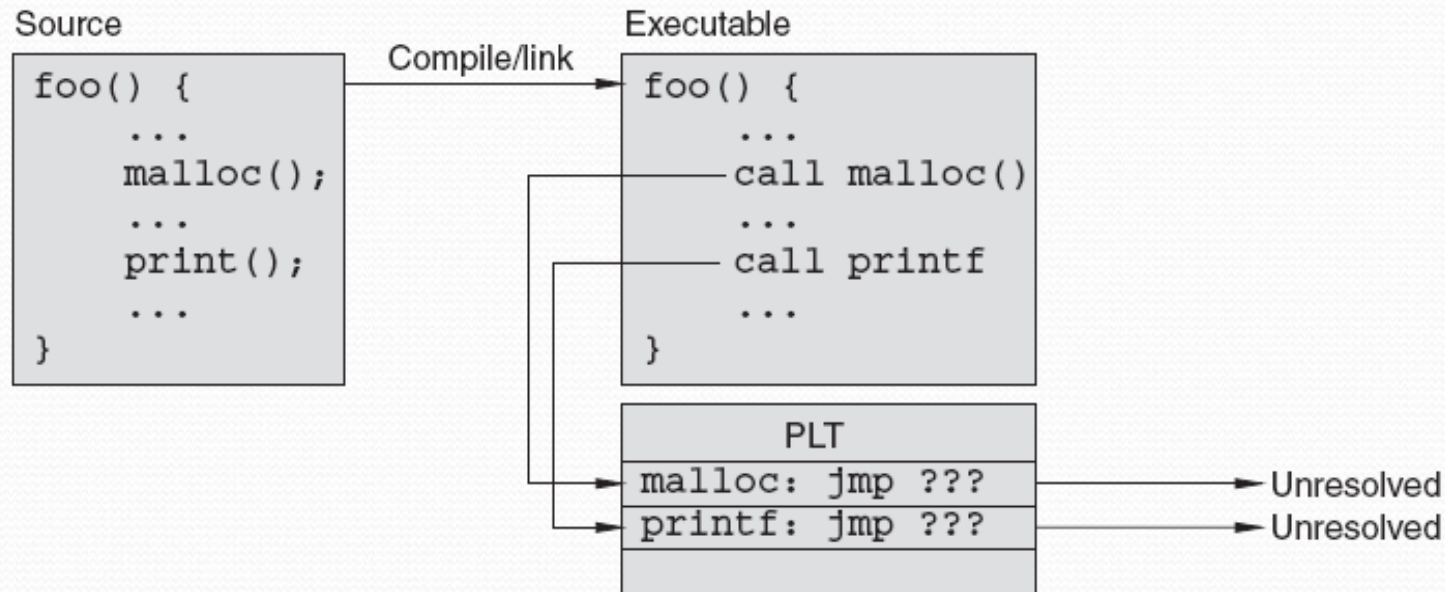
- El código de la biblioteca esta en todos los ejecutables que la usan, lo que desperdicia disco y memoria.
- Si actualizamos las bibliotecas, debemos recompilar el programa para que se beneficie de la nueva versión.

Las bibliotecas dinámicas se integran en ejecución, para ello se ha realizado la reubicación de módulos. Su diferencia con un ejecutable: tienen tabla de símbolos, información de reubicación y no tiene punto de entrada.

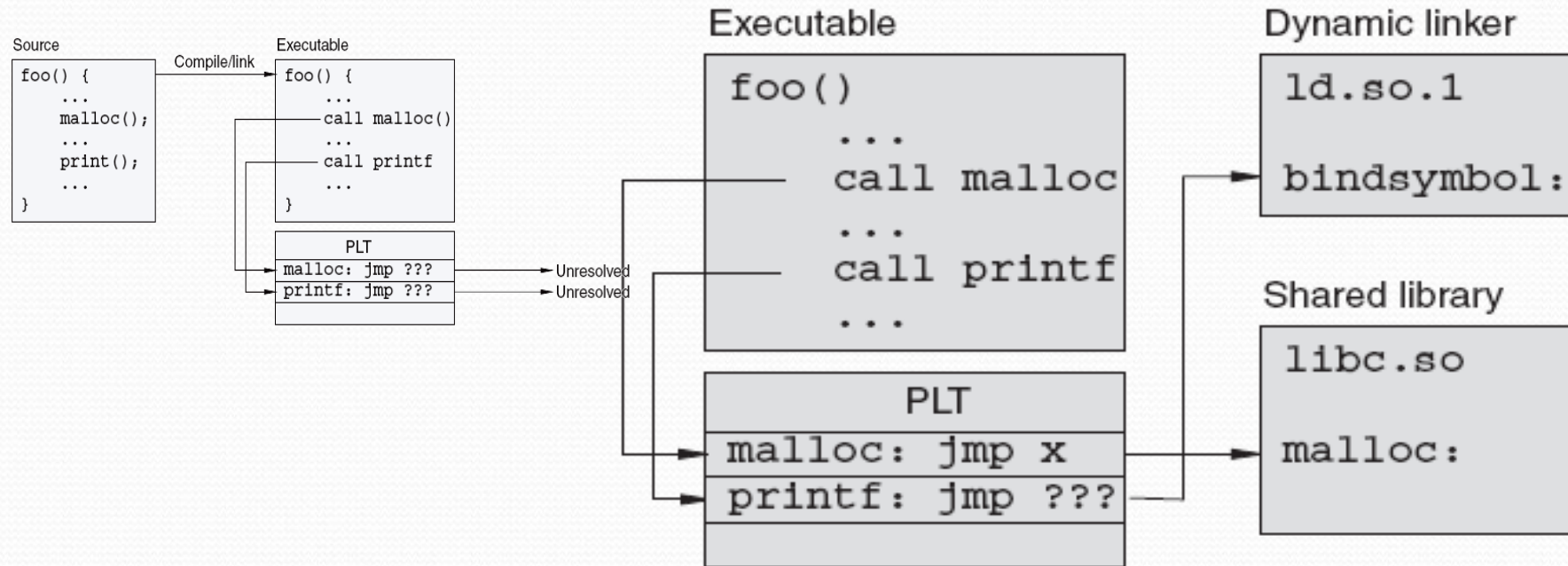
Pueden ser:

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

Estructura de un ejecutable tras el proceso de compilación y enlazado (1/2)



Estructura de un ejecutable tras el proceso de compilación y enlazado (2/2)



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 ver las bibliotecas enlazadas con un programa:

```
ldd hola
```

3.8 Automatización del Proceso de Compilación y Enlazado. Herramientas y Entornos

Automatización en la Construcción de Software

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**
- IDE (Integrated Development Environment), que embebe los guiones y el proceso de compilación y enlazado.