3.1 Lenguajes de Programación.

Concepto de lenguaje de programación

Un lenguaje de programación es un conjunto de símbolos y reglas para combinarlos, que se usan para expresar algoritmos. Los lenguajes de programación (al igual que los lenguajes que usamos para comunicarnos) poseen:

- Un **léxico**: vocabulario o conjunto de símbolos permitidos
- Una **sintaxis**: indica cómo realizar construcciones del lenguaje
- Una **semántica**: determina el significado de cada construcción correcta

Están específicamente diseñados para programar ordenadores. Sus principales características son:

- **Independientes** de la arquitectura física del computador. No obligan al programador a conocer los detalles del computador que utiliza, y permiten utilizar los mismos programas en computadores diferentes, con distinto lenguaje máquina (portabilidad)
- Una **sentencia** en un lenguaje de alto nivel da lugar, al ser traducida, 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

- Utilizan **notaciones cercanas** a las habituales en el ámbito en que se usan. Sentencias o frases muy parecidas al lenguaje matemático o al lenguaje natural (palabras o términos en inglés).

Lenguajes de primera generación:

- Lenguaje máquina: Esta escrito en binario, es lo que entiende el ordenador de forma directa
- Ensamblador: Símbolos que son traducidos al binario

Lenguajes de segunda generación:

- Macro ensamblador: Son secuencias de instrucciones que se empaquetaban todas juntas formándose una macro. Consta del repertorio y de las distintas macros. Se traduce de igual manera que el ensamblador. Tiene una serie de instrucciones y cierta estructura.

Lenguajes de tercera generación: Los primeros lenguajes que intentaban alejarse del ensamblador eran extremadamente ineficientes en el proceso de traducción, se generaba mucho código basura que hacía que fuesen un desastre. Este paso al lenguaje de tercera generación es un desarrollo un poco mayor del macro ensamblador.

- FORTRAN: Desarrollado por IBM, fue el primer lenguaje con un compilador eficiente que hacía igual de eficaz el código máquina al código escrito.

Los **lenguajes de alto nivel** son lenguajes que combinan símbolos y estructura. La estructura a su vez está compuesta por datos e instrucciones.

La utilización de conceptos habituales suele implicar, las siguientes cualidades:

- Instrucciones expresadas por medio de *texto* (con caract. alfanuméricos y especiales)
- Se puede asignar un *nombre simbólico* a determinados componentes del programa. El programador puede definir las variables que desee

- Dispone de instrucciones potentes, conteniendo *operadores y funciones* (aritméticas, especiales, lógicas, de tratamiento de caracteres, etc.)
- Pueden incluirse *comentarios*, lo que les da legibilidad a los programas

Como consecuencia de este alejamiento de la máquina y acercamiento a las personas, los programas escritos en lenguajes de programación no pueden ser directamente interpretados por el ordenador, siendo necesario realizar previamente su traducción a lenguaje máquina. Hay dos tipos de traductores de lenguajes de programación: los compiladores y los intérpretes.

En los lenguajes de alto nivel aparece la idea de **tipo de dato** la cual se formalizó matemáticamente.

Un tipo de dato está compuesto por una terna (G, O, P). Donde:

- G: el género del tipo conjunto de valores que puede tomar dentro de ese tipo (Es la representación)
- O: indica las operaciones que soporta el tipo
- P: representa las propiedades de esas operaciones.

Ejemplo INT: $\begin{cases} G = \{-2^{31}, \dots, 2^{31}\} \\ O = \text{Operaciones} \\ P = \text{Propiedades} \end{cases}$

O y P conforman un grupo conmutativo.

Existen tres tipos de datos:

- **Primitivos**: No tienen estructura son los enteros, reales, caracteres etc.
- **Estructurados**: Los datos estructurados necesitan tener definido de forma anterior el elemento G, por ejemplo, en un array necesitamos decir en número filas y el número columnas y lo que hay dentro.
- Tipos de datos **abstractos**: Nosotros definimos la terna GOP.

Las **instrucciones** son las distintas órdenes que se le dan al ordenador desde el lenguaje de programación. Con el proceso de desarrollo de los lenguajes de programación se incorporan junto con las expresiones el control de flujo el cual regula el orden en el que se ejecutan las instrucciones.

Las **expresiones** son las operaciones como: a + b + c, d /e…

Las **sentencias** son más complejas. Entre ellas están las rupturas de flujo, las lecturas, escrituras, asignaciones etc.

Desde el punto de vista de la traducción con las estructuras de alto nivel aparecen dos tipos de sentencias:

- Sentencias semánticamente simples: Son macros entre ellas están las asignaciones y las operaciones de E/S de una sola variable. No se tiene que expresar respecto a las demás
- Sentencias semánticamente compuestas: Son todas las demás sentencias.

3.2 Construcción de Traductores.

Definición de traductor

Como el computador puede interpretar y ejecutar únicamente código máquina, existen programas traductores, que traducen programas escritos en lenguajes de programación a lenguaje máquina.

Un **traductor** es un programa que recibe como entrada un texto en un lenguaje de programación concreto, y produce, como salida, un texto en lenguaje maquina equivalente.

El programa inicial se llama **programa fuente** (lenguaje fuente, define a una máquina virtual) y el programa obtenido, **programa objeto** (lenguaje objeto, define a una máquina real).

Definición de compilador

Un **compilador** traduce un programa fuente, escrito en un lenguaje de alto nivel, a un programa objeto, escrito en lenguaje ensamblador o máquina. El compilador traduce la especificación de entrada a lenguaje máquina incompleto y con instrucciones maquinas incompletas, por lo que necesita de un complemento llamado enlazador.

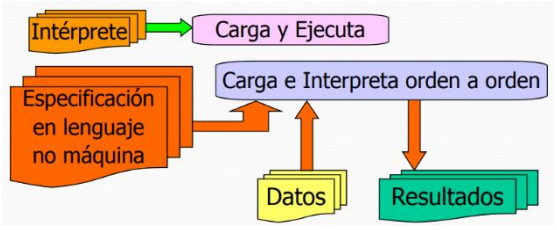
El programa fuente suele estar contenido en un archivo y el programa objeto puede almacenarse como archivo en memoria masiva para ser procesado posteriormente, sin necesidad de volver a realizar la traducción. Una vez traducido el programa, su ejecución es independiente del compilador.

Un **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.

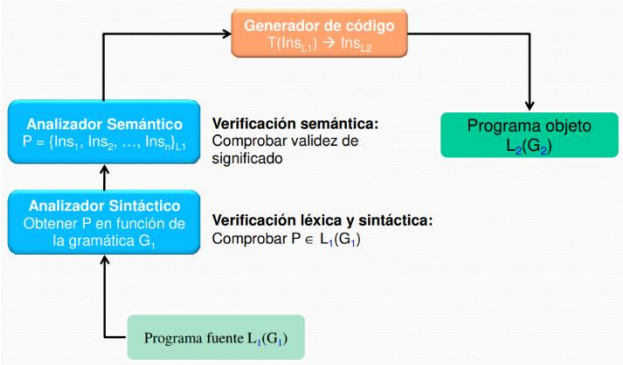
Definicion de intérprete

Un **intérprete** hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador. El intérprete capta una sentencia fuente, la analiza e interpreta dando lugar a su ejecución inmediata, no creándose un archivo o programa objeto almacenable en memoria masiva para posteriores ejecuciones.

Un 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.



Esquema de traducción



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:

- $\boldsymbol{V_N}$ es el conjunto de símbolos no terminales. Tiene estructura interna. Son variables sintácticas que denotan conjuntos de cadenas. Las instrucciones y sentencias son no terminales. Los no terminales imponen una estructura jerárquica sobre el lenguaje.
- $\boldsymbol{V_T}$ es el conjunto de símbolos terminales. No tienen estructura interna. El nombre del token es un sinónimo de terminal.
- $\boldsymbol{P}$ es el conjunto de producciones. Son las reglas de producción. Las que regulan la composición entre los elementos de la gramática. Cada producción cosiste en un no terminal conocido como encabezado, un símbolo y un cuerpo o lado derecho que consiste en cero o más terminales y no terminales, los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado
- $\boldsymbol{S}$ es el símbolo inicial. Un no terminal se distingue como el símbolo inicial, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención las producciones para el símbolo inicial se listan primero

Ejemplo

Dada la gramática siguiente:

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

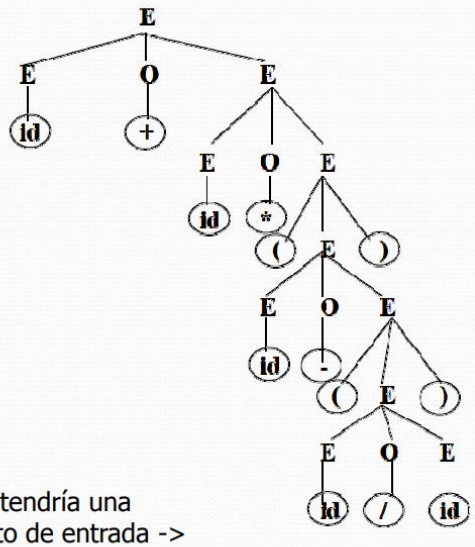
$V_N = \{E,O\}$

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

$S = E$

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.



Tipos de gramáticas

- **Gramáticas regulares/expresiones regulares:** Son las gramáticas más básicas que solo leen expresiones /ejemplo anterior.
- **Gramática independiente del contexto**: Así son la de la mayoría de los lenguajes de programación. Consisten en terminales, no terminales, un símbolo inicial y producciones
- **Gramáticas sensibles al contexto**: Entre estas se incluyen las gramáticas de dos niveles y la gramática con atributos.
- **Gramáticas libres**

Todas estas gramáticas se definen igual, en lo que varían es en sus reglas de producción.

Una **gramática con atributo**s es una cuádrupla compuesta por una gramática independiente del contexto, un conjunto de atributos (son asociados a los símbolos de la gramática inicial) , un conjunto de reglas de valor y condiciones sobre los valores de los atributos.

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

El lexema es la unidad mínima de significado, es un símbolo terminal. Todos los caracteres que pongamos o vayamos a poner deben de estar en la gramática.

El análisis sintáctico coge los identificadores (símbolos terminales) **recogidos por el análisis léxico** u analiza la expresión de los diversos identificadores.

3.3 Proceso de Compilación

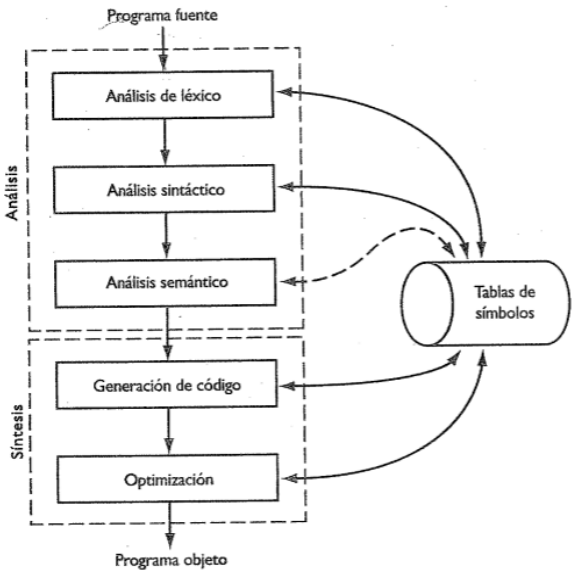


Figura 14.3. *Fases en la traducción de un programa fuente a un programa objeto, por un compilador.*

La traducción por un compilador (la **compilación**) consta de dos etapas fundamentales, que a veces no están claramente diferenciadas a lo largo del proceso:

- Análisis del programa fuente
- Síntesis del programa objeto

La compilación es un proceso complejo y que consume a veces un tiempo muy superior a la propia ejecución del programa. En cualquiera de las fases de análisis el compilador puede dar mensajes sobre los errores que detecta en el programa fuente, cancelando en ocasiones la compilación para que el usuario realice las correcciones oportunas en el archivo fuente.

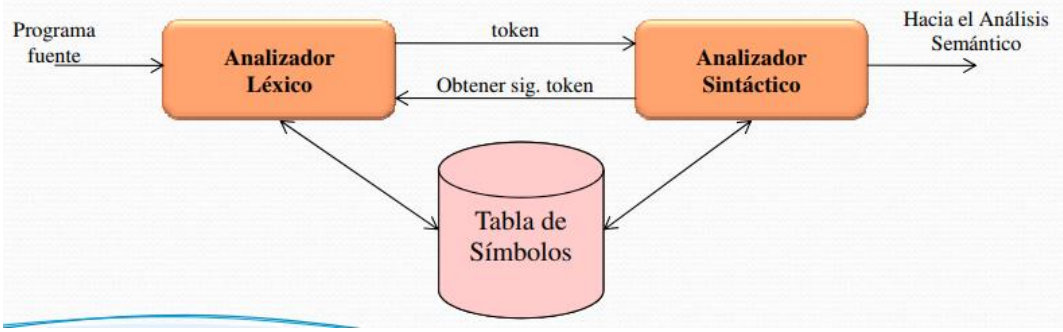
3.3.1 Análisis Léxico.

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. Consiste en descomponer el programa fuente en sus elementos constituyentes o símbolos de léxico (tokens).

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.



El analizador léxico realiza otras funciones secundarias:

- Identifica, y salta, comentarios
- Identifica, y pasa, espacios en blanco y tabulaciones
- Informa sobre posibles errores de léxico

Ejemplo 1

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

Ejemplo 2

$$
\begin{array}{rcl}
S & \rightarrow & A \mid C \\
A & \rightarrow & \mathrm{id} = E \\
C & \rightarrow & \mathrm{if}\ E\ \mathrm{then}\ S \\
E & \rightarrow & E\ O\ E \mid (E) \mid \mathrm{id} \\
O & \rightarrow & + \mid - \mid * \mid / \\
\mathrm{id} & \rightarrow & \mathrm{letra} \mid \mathrm{id}\ \mathrm{digito} \mid \mathrm{id}\ \mathrm{letra} \\
\mathrm{letra} & \rightarrow & a \mid b \mid \cdots \mid z \\
\mathrm{digito} & \rightarrow & 0 \mid 1 \mid \cdots \mid 9
\end{array}
$$

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

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)

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 más veces, operador *
2. Una 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 manera: $a|b|c|\ldots|z = [a-z]$

Especificación de los tokens

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

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

El compilador comprueba cada símbolo según la gramática hasta que termina la expresión, se lee de izquierda a derecha hasta que se localiza un separador.

Conforme el analizador va leyendo el programa construye una tabla de símbolos con las declaraciones de variable. Esta tabla incluye el lexema, el tipo, la longitud y la dirección de memoria asociada. Lo primero que aparece en un programa es la zona de declaraciones y las bibliotecas, estas son apuntadas en la tabla de símbolos con las correspondientes direcciones asociadas.

3.3.2 Análisis Sintáctico

La sintaxis de un lenguaje de programación especifica cómo deben escribirse los programas, mediante un conjunto de reglas de sintaxis o gramática del lenguaje. Un programa es sintácticamente correcto cuando sus estructuras (expresiones, sentencias, asignaciones…) aparecen en un orden correcto. Una construcción puede ser sintácticamente correcta y carecer de sentido.

Se han definido varios sistemas para definir la sintaxis de los lenguajes de programación (metalenguajes), como por ejemplo la notación BNF y los diagramas sintácticos.

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

Los lenguajes están hechos de tal manera que una vez localizado el primer token ya se sabe cuál va a ser su expresión. Dependiendo del tipo de gramática se va construyendo las expresiones de un modo u otro.

- Las gramáticas de tipo **LL** (left, left) construyen de arriba a abajo y de izquierda a derecha. Este árbol es compuesto por el análisis sintáctico.
- Las familias de gramáticas de tipo **LR** (left, right) (son más complejas y detectan más elementos). El árbol es ascendente y va de derecha a izquierda.

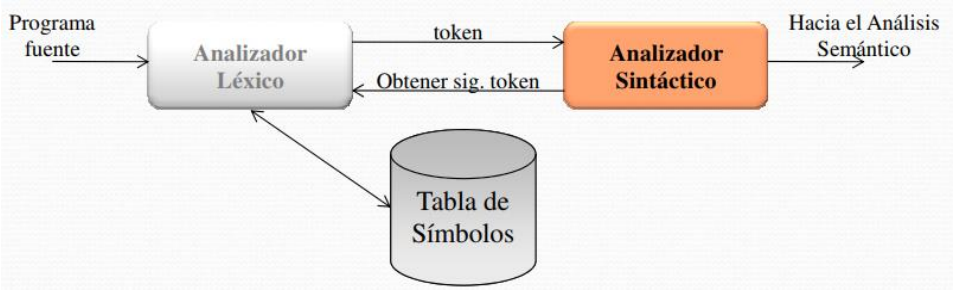
Cualquier sentencia reconoce los identificadores y crea la tabla de símbolos.

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

Libro Aho (lioso)

Si utilizamos una variable sintáctica *instr* para denotar las instrucciones, y una variable *expr* para denotar las expresiones, la siguiente producción especifica la estructura de esta forma de instrucción condicional:

```
instr -> if (expr) instr else instr
```

La gramática libre de contexto consiste en terminales, no terminales, un símbolo inicial y producciones:

1. Los **terminales** son los símbolos básicos a partir de los cuales se forman las cadenas. El término *nombre de token* es un sinónimo de terminal. Las terminales son los primeros componentes de los tokens que produce el analizador léxico. En el ejemplo, los terminales son las palabras reservadas *if* y *else*, y los símbolos (y).
2. Los **no terminales** son variables sintácticas que denotan conjuntos de cadenas. En el ejemplo *instr* y *expr* son no terminales. Los conjuntos de cadenas denotados por los no

terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.

3. En una gramática, un no terminal se distingue como el **símbolo inicial**, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Las producciones para el símbolo inicial se listan primero.
4. Las producciones de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada producción consiste en:
   - **a.** Un no terminal, conocido como *encabezado* o *lado izquierdo* de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.
   - **b.** El símbolo $\rightarrow$ o $:=$.
   - **c.** Un *cuerpo* o *lado derecho*, que consiste en cero más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

Ejemplo. En esta gramática, los símbolos de los terminales son: `id + - * / ( )`

Los símbolos de los no terminales son *expresión*, *term* y *factor*, y *expresión* es el símbolo inicial.

Gramática para las expresiones aritméticas simples:

expresión → expresión + term
expresión → expresión - term
expresión → term
term → term * factor
term → term / factor
term → factor
factor → (expresión)
factor → id

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

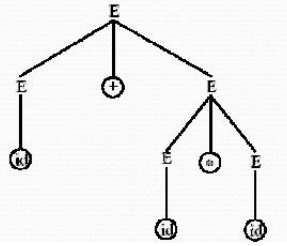
se dice que es una gramática libre de contexto cuando el conjunto de producciones P obedece al formato: $P = \{A \rightarrow \alpha \ / A \in V, \ \alpha \subset (V_N \cup V_T)^*\}$

es decir, solo 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.

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.

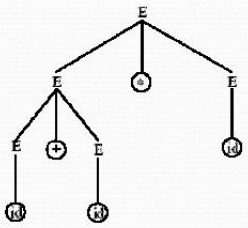


$E \rightarrow E+E$

$id+E$

$id+E*E$

$id+id*id$



$E \rightarrow E*E$

$E*id$

$E+E*id$

$id+id*id$

3.3.3 Análisis Semántico.

La semántica de un lenguaje de programación es el significado dado a las distintas construcciones sintácticas. El proceso de traducción es la generación de un código en lenguaje máquina con el mismo significado que el código fuente. Es el análisis que comprueba la lógica de lo que hemos escrito. El significado está ligado a la estructura sintáctica de las sentencias. En el proceso de traducción, el significado de las sentencias se obtiene de la identificación sintáctica de las construcciones sintácticas y de la información almacenada en las tablas de símbolos.

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**.

- 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).

3.3.4 Generación y Optimización de Código.

Generación de código

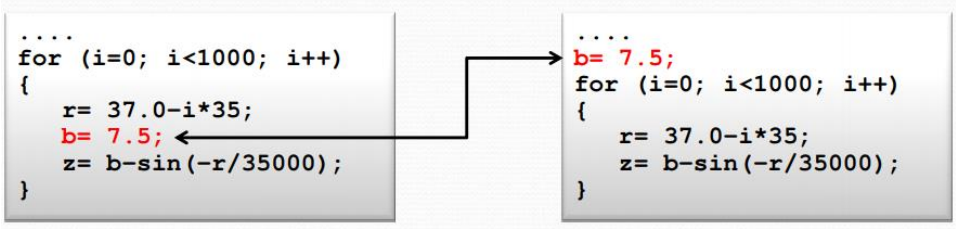
- Durante esta fase se crea el archivo con un código en lenguaje objeto con el mismo significado que el texto fuente. En la mayoría de los casos, se traduce a un lenguaje intermedio con el propósito de optimizar el código posteriormente. En la generación de código intermedio se completan y consultan las tablas generadas en fases anteriores (tablas de símbolos, de constantes,etc.). También se realiza la asignación de memoria a los datos definidos en el programa.

- El archivo-objeto generado puede ser (dependiendo del compilador) directamente ejecutable, o necesitar otros pasos previos a la ejecución (ensamblado, encadenado y carga).
- Una vez pasados los distintos analizadores se descomponen los while,for etc y se pasan las expresiones complejas a otras más simples. Este código intermedio es descompuesto en bloques básicos. Los bloques básicos son "cada linea" del programa. El compilador va optimizando los distintos bloques básicos según la lógica, a partir de ahí se genera un fichero en cuartetos organizados del cual se generará el código máquina.
- 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).
- La generación de código puede realizarse añadiendo procedimientos en determinados puntos del proceso de análisis, que generen las instrucciones en lenguaje objeto equivalentes a cada construcción en lenguaje fuente reconocida.
- En la generación de código intermedio se completan y consultan las tablas generadas en fases anteriores (tablas de símbolos, de constantes...). También se realiza la asignación de memoria a los datos definidos en el programa

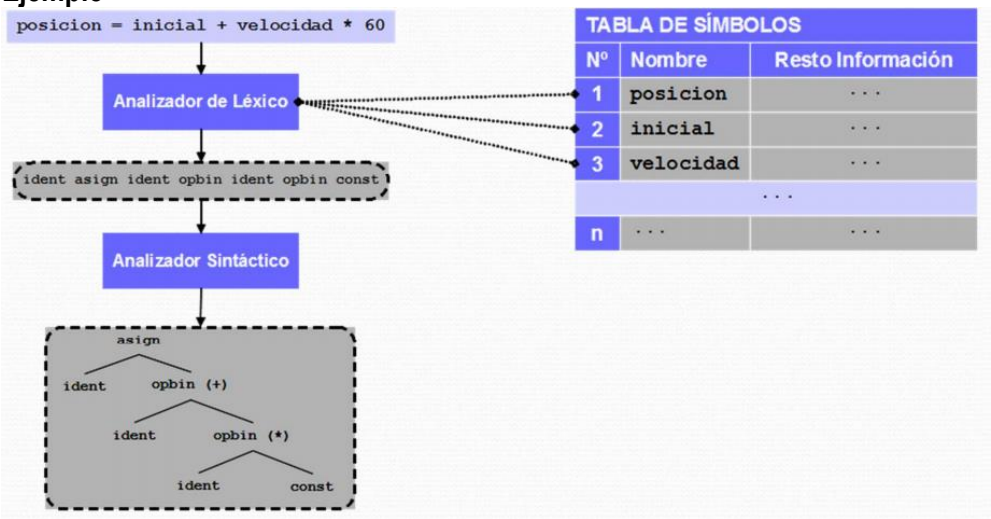
Optimización de código

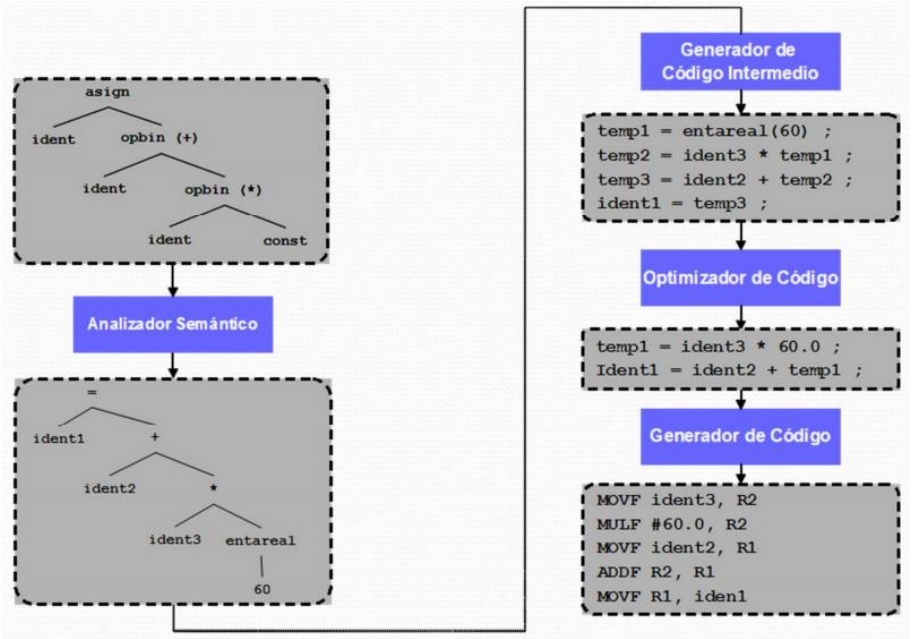
- Esta fase existe para mejorar el código mediante comprobaciones locales a un grupo de instrucciones (bloque básico) o a nivel global
- Se pueden realizar optimizaciones de código tanto al código intermedio (si existe) como al código objeto final. Generalmente, las optimizaciones se aplican a códigos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

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



Ejemplo





3.4 Intérpretes.

Intérprete: hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador. El intérprete capta una sentencia fuente, la analiza e interpreta dando lugar a su ejecución inmediata.

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

En la práctica, el usuario crea un archivo con el programa fuente. Esto suele realizarse con un editor específico del propio intérprete del lenguaje. Según se van almacenando las instrucciones simbólicas, se analizan y se producen los mensajes de error correspondientes; así el usuario puede dar la orden de ejecución y el intérprete lo ejecuta línea a línea. Siempre el análisis antecede inmediatamente a la ejecución.

¿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 tienen 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.

3.5 Modelos de Memoria de un Proceso.

Modelos de memoria de un proceso

El SO gestiona el mapa de memoria de un proceso durante la vida del mismo. Además, el mapa inicial de un proceso está muy vinculado con el archivo que contiene el programa ejecutable asociado al mismo.

Los programadores desarrollan sus aplicaciones utilizando lenguajes de alto nivel. Por lo que una aplicación estará compuesta por un conjunto de módulos de código fuente que deberán ser procesados para obtener el ejecutable de la aplicación. Este procesamiento consta de dos fases:

- Compilación: se genera el código máquina correspondiente a cada módulo fuente de la aplicación asignando direcciones a los símbolos definidos en el módulo y resolviendo las referencias a los mismos. Como resultado de esta fase se genera un módulo objeto por cada archivo fuente.
- Montaje o enlace: se genera un ejecutable agrupando todos los archivos objeto y resolviendo las referencias entre módulos.

En resumen, los elementos responsables de la gestión de memoria son:

- Lenguaje de programación
- Compilador
- Enlazador
- SO
- MMU (Memory Management Unit)

Niveles de la gestión de memoria

- Nivel de **procesos:** determina cómo se reparte el espacio de memoria entre los procesos existentes. Nivel gestionado por el SO. Operaciones:
  - Crear el mapa de memoria del proceso (*crear_mapa*)
  - Eliminar el mapa de memoria del proceso (*eliminar_mapa*)
  - Duplicar el mapa de memoria del proceso (*duplicar_mapa*)
  - Cambiar de mapa de memoria de proceso (*cambiar_mapa*)
- Nivel de **regiones**: establece cómo se reparte el espacio asignado al proceso entre las regiones del mismo. Nivel manejado por el SO. Operaciones:
  - Crear una región dentro del mapa de un proceso (*crear_región*)
  - Eliminar una región del mapa de un proceso (*eliminar_región*)
  - Cambiar el tamaño de una región (*redimensionar_región*)
  - Duplicar una región del mapa de un proceso en el mapa de otro (*duplicar_región*)
- Nivel de **zonas:** reparto de una región entre las diferentes zonas (nivel estático, dinámico basado en pila y dinámico basado en *heap*) de ésta. Gestionado por el lenguaje de programación con soporte del SO. Operaciones:
  - Reservar una zona (*reservar_zona*)
  - Liberar una zona reservada (*liberar_zona*)
  - Cambiar el tamaño de una zona reservada (*redimensionar_zona*)

Necesidades de memoria de un proceso (necesidades de los programas)

- Tener un espacio lógico propio e independiente
- Estar protegido de otros programas que estén ejecutándose simultáneamente
- Compartir memoria con otros procesos para poder comunicarse eficientemente
- Tener soporte para sus distintas regiones
- Tener facilidades para su depuración
- Poder usar un mapa de memoria muy grande, si lo precisa
- Utilizar distintos tipos de objetos de memoria
- Poder hacer persistentes los datos que así lo requieran
- Desarrollarse de una forma modular
- Poder realizar la carga dinámica de módulos en tiempo de ejecución
- Espacios lógicos independientes para los programas

Tipos de datos

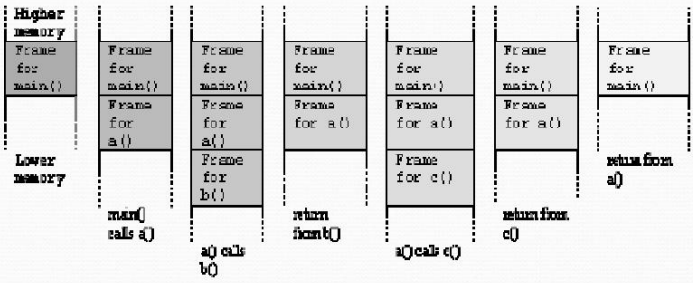
- **Datos estáticos**: datos que existen durante toda la vida del programa. Cada dato tiene asociada una posición fija en el mapa de memoria del proceso durante toda la ejecución del programa. Tipos:
    - **Globales**: a todo el programa, a todo un módulo o locales a una función, dependiendo del ámbito de visibilidad de la variable correspondiente. Es muy importante para el compilador y montador.
    - **Constantes o variables**: las constantes no se modifican. El compilador puede hacer ciertas comprobaciones y generar un error en caso de que se intente modificar. Pero, es necesario asegurarse en tiempo de ejecución de que no se puede modificar, para lo cual habrá que asociar este tipo de dato a una región que no pueda modificarse.
    - **Con o sin valor inicial** – direccionamiento relativo: PIC. En caso de que tenga un valor asociado, habrá que asegurarse de que este esté almacenado en la memoria cuando el proceso intente acceder al mismo. Dado que el mapa inicial del proceso se construye a partir del ejecutable, habrá que almacenar ese valor en el mismo. El código PIC se trata de una región privada ya que cada proceso que ejecuta un determinado programa necesita una copia propia de las variables del mismo.
- **Datos dinámicos asociados a la ejecución de una función**: la vida de este tipo de objetos de memoria está asociada a la ejecución de una función y se corresponden con las variables locales (las no estáticas) y los parámetros de la función. Se crean cuando se invoca la función correspondiente y se destruyen cuando termina la llamada. En consecuencia, estas variables no tienen asignado espacio en el mapa inicial del proceso ni en el ejecutable. Se almacenan dinámicamente en la pila del proceso, en una estructura de datos denominada **registro de activación**, donde se guardan las variables locales, parámetros y la información necesaria para retornar al punto de llamada cuando termina la ejecución de la función. La dirección que corresponde a una variable de este tipo se determina en tiempo de ejecución. Para acceder a esta variable, hay que consultar el último registro de activación apilado. Este tipo de variables, al igual que las estáticas, pueden tener asignado un valor inicial.

- **Datos dinámicos controlados por el programa – heap**. Se trata de datos dinámicos que el programa crea cuando considera oportuno, usando los mecanismos proporcionados por el lenguaje de programación correspondiente. Los datos se guardan en la región heap, donde se van almacenando todos los datos dinámicos usados por el programa. El espacio asignado se liberará cuando ya no sea necesario, o bien porque así lo indique el programa usando un mecanismo de detección automático, denominado **recolección de basura**.

La dirección asociada a un dato de este tipo solo se conoce en tiempo de ejecución, cuando el entorno de ejecución del lenguaje le asigna una zona del heap. Las referencias a este tipo de objetos no necesitan reubicarse y cumplen la propiedad PIC. Por tanto, el compilador y las bibliotecas del lenguaje resuelven todos los aspectos de implementación requeridos.

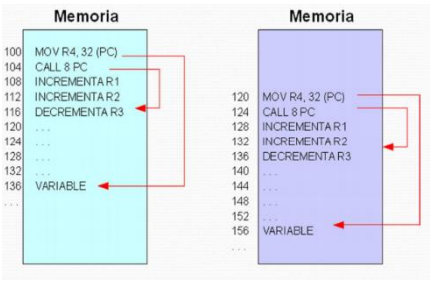
Ejemplo de evolución de la Pila (Stack) en la ejecución de un programa

```
#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_dinanica.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

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

Una vez que los programadores terminan la codificación de un programa, este debe pasar por varias fases hasta que pueda ejecutarse

1. **Compilación**

Habitualmente, los programadores desarrollan sus aplicaciones utilizando lenguajes de alto nivel, dividiendo la funcionalidad en múltiples módulos (ficheros de código fuente), para facilitar un desarrollo incremental y la reutilización del código. El compilador se encarga de procesar de forma independiente cada fichero, generando el código objeto correspondiente. El compilador se encarga de realizar las siguientes operaciones:

- Genera el código objeto pertinente y determina cuanto espacio ocupan los datos estáticos, distinguiendo entre:
  - Carácter constante
  - Carácter variable
    - Tienen un valor inicial asignado
    - No tienen un valor inicial asignado

  El compilador estructura toda esta información en **secciones**, que son sus unidades de organización
- Asigna direcciones consecutivas a los símbolos estáticos; primero a las constantes, luego a las variables con valor inicial y, por último, a las variables sin valor inicial.
- Resuelve las referencias a los símbolos estáticos, tanto los correspondientes a datos como a instrucciones. Esas referencias pueden resolverse con un direccionamiento absoluto (reubicar) o con uno relativo a PC.
- Las referencias a símbolos dinámicos se resuelven utilizando un redireccionamiento relativo a la pila (datos asociados a la invocación de una función) o redireccionamiento indirecto a través de un puntero (variable del *heap*)

El compilador también genera:

- Una sección que almacena información de que instrucciones precisan reubicación o están pendientes de resolver se referencia a un símbolo externo
- Una sección que contiene una tabla de símbolos
- Una sección que incluye información de depuración

Como resultado de la compilación de un módulo, se genera un **fichero objeto** cuya estructura habitual incluye una **cabecera** (información de control que permite interpretar el contenido del ejecutable) y una serie de **secciones** (código, datos con valor inicial, datos sin valor inicial, información de reubicación, tabla de símbolos e información de depuración).

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
```

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
```

2. **Montaje o enlace**

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.

Tipos de enlazado y ámbito

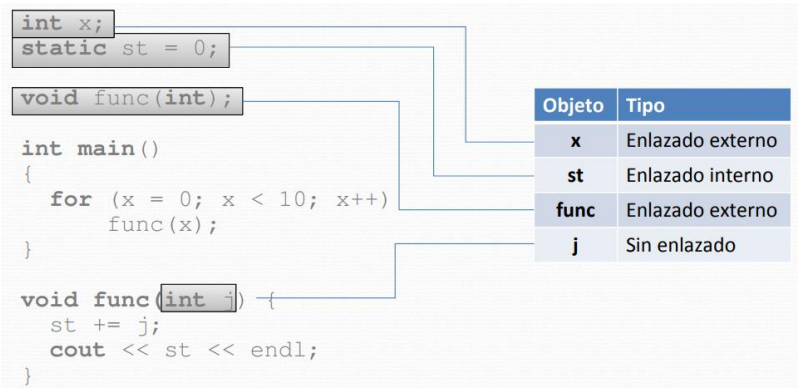
- Atributos de enlazado: externo, interno o sin enlazado
- Los tipos de enlazado definen una especie de ámbito:
  - Enlazado externo --> visibilidad global

- Enlazado interno --> visibilidad de fichero
- Sin enlazado --> visibilidad de bloque

Reglas de enlazado

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

Ejemplo



3. **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.

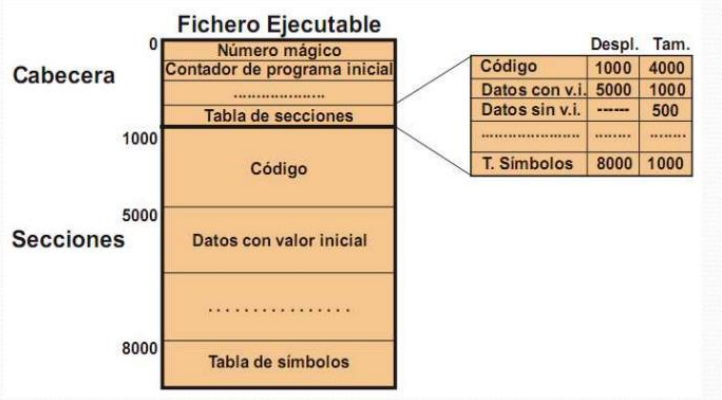
Diferencias entre archivos objeto y archivos ejecutables

Los archivos objeto (resultado de la compilación) y ejecutable (resultado del enlazado) son muy similares en cuanto a contenidos.

Sus 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 archivos objeto y ejecutables

	Descripción
a.out	Es el formato original de los sistemas Unix. Consta de tres secciones: `text, data y bss,` que se corresponden con el código, datos inicializados y sin inicializar. No tiene información para depuración
COFF	El *Common Object File Format* posee múltiples secciones, cada una con su cabecera, pero están limitadas en número. Aunque permite información de depuración, esta es limitada. Es el formato utilizado por Windows.
ELF	*Executable and Linking Format* es similar al COFF, pero elimina alguna 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.

3.7 Bibliotecas.

Definiciones

- Una **biblioteca** es una colección de objetos normalmente relacionados entre sí. En el sistema existe un conjunto de bibliotecas predefinidas que proporcionan servicios a las aplicaciones. Estos incluyen desde las API del lenguaje hasta las API del sistema operativo. Además, hay más bibliotecas para las distintas aplicaciones.
- Cualquier usuario puede crear su propia biblioteca, para organizar mejor los módulos de una aplicación y facilitar que las aplicaciones compartan módulos.
- Los módulos del programa pueden incluir referencias a símbolos definidos en algunos de los objetos de una determinada biblioteca, tanto a funciones como a variables exportadas por la misma. En la etapa de resolución de símbolos, si una determinada referencia no se resuelve usando los objetos que forman parte de la aplicación, el montador busca el símbolo en cada una de las bibliotecas especificadas por el usuario en el mandato de montaje, siguiendo el orden especificado en el mismo. Una vez encontrado el símbolo en una determinada biblioteca, el montador extraerá ese objeto de la biblioteca y lo incorporará junto con el resto de los módulos objetos de la aplicación.
- 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)

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
```

Que el programa contenga todo el código que necesita el programa para poder ejecutarse tiene varios **inconvenientes** como:

- El fichero ejecutable puede ser bastante grande
- Todo programa en el sistema que use una determinada función de biblioteca tendrá una copia del código de la misma.
- Si se están ejecutando simultáneamente varias aplicaciones que usan una misma función de biblioteca existirán en memoria múltiples copias del código de dicha función, aumentando el gasto de memoria
- La actualización de una biblioteca implica tener que volver a generar los ejecutables que la incluyen, por muy pequeño que sea el cambio que se ha realizado sobre la misma.

Para resolver estas deficiencias se usan las bibliotecas montadas dinámicamente llamadas simplemente bibliotecas dinámicas.

Bibliotecas dinámicas

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:

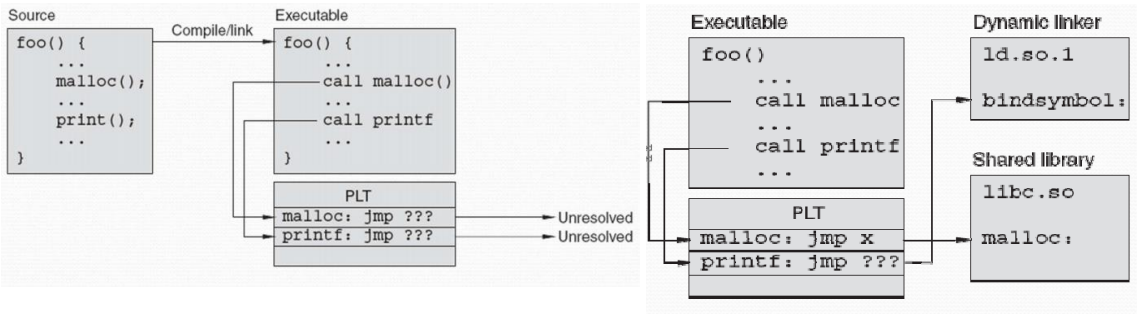
- Bibliotecas compartidas de carga dinámica – la reubicación se realiza en tiempo de enlazado
- Bibliotecas compartidas enlazadas dinámicamente – el enlazado se realiza en ejecución.

El montaje de las bibliotecas dinámicas se hace en tiempo de ejecución del programa. La biblioteca dinámica la genera el montador, de manera similar a lo que ocurre con un ejecutable. La principal diferencia entre el ejecutable y la biblioteca dinámica consiste en que la biblioteca dinámica contendrá información de reubicación y una tabla de símbolos, puesto que en tiempo de ejecución hay que realizar la tabla de resolución de símbolos y la reubicación de regiones.

Una vez creada la biblioteca dinámica esta puede ser utilizada por cualquier programa. Cuando en la fase de montaje de un programa, en la etapa de resolución de símbolos se detecta que uno de ellos está definido en una de las bibliotecas dinámicas especificadas por el usuario no se lleva a cabo la resolución ni se incluye en el ejecutable que contiene la lista de bibliotecas dinámicas que requiere el programa durante la ejecución. Como parte del proceso de montaje se incluye un módulo de montaje dinámico que se encargará de realizar en tiempo de ejecución la carga y el montaje de las bibliotecas dinámicas usadas por el programa.

La implementación de estas bibliotecas tiene problemas como que las referencias incluidas en la biblioteca deben ajustarse para que estén acordes con la zona del mapa del proceso. También las referencias del programa a los símbolos de la biblioteca y las de la propia biblioteca a los símbolos de otras bibliotecas anteriormente cargadas y montadas, deben resolverse y ajustarse de acuerdo con su ubicación en el mapa del proceso.

Estructura de un ejecutable tras el proceso de compilación y enlazado



Creación y uso de las bibliotecas dinámicas

- Generamos el objeto de la biblioteca:
  `gcc -c -fPIC calc.mean_c -o calc_mean.o`
- Generamos 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`

Automatizacion en la construcción del 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.