TEMA3-FSoftware.pdf

Airin86

Fundamentos del Software

1º Doble Grado en Ingeniería Informática y Matemáticas

Facultad de Ciencias
Universidad de Granada

Fundamentos del Software Irene Pérez Martínez

TEMA 3: Compilación y enlazado de programas

Los humanos somos incapaces de escribir qué queremos que haga un ordenador usando únicamente 0s y 1s: nos volveríamos locos. Por eso necesitamos un lenguaje para programar que sea cómodo para nosotros, pero... ¿eso lo entiende un ordenador? ¿Y si hay errores en el código? ¿Cómo obtenemos un archivo ejecutable de lo que hemos programado?

En este tema vamos a ir viendo, paso por paso, cómo es el proceso de crear un archivo ejecutable. Partimos desde que creamos un programa con lenguaje de programación, pasando por la traducción y evaluación del mismo, hasta que conseguimos un archivo que el ordenador entiende y puede ejecutar.

1. Lenguajes de programación

El **lenguaje de programación** es un conjunto de símbolos y de reglas para combinarlos, que se usan para expresar algoritmos. Al igual que para comunicarnos entre nosotros, para programar necesitamos un lenguaje especial que permita relacionarnos con el ordenador; de ahí nace la necesidad de crear los lenguajes de programación. Hay múltiples lenguajes diferentes para programar, como C++ o Java, que poseen diferentes características, funcionalidades u objetivos. Aún así, todos comparten las siguientes características:

- **Independencia** de la arquitectura del computador, es decir, del hardware. Esto permite que los programas sean **portables**; puedes haberlo creado en un computador y ejecutarlo en otro. Da libertad al programa.
- Una **instrucción** en un lenguaje de alto nivel da lugar, tras el proceso de traducción, a **varias instrucciones** en lenguaje máquina. Esto permite simplificar la realización de programas, ya que el programador no tiene que redactar tanto.
- Un lenguaje de alto nivel utiliza **notaciones fácilmente reconocibles** por las personas en el ámbito en que se usan. Esto significa que, en vez de usar lenguaje ensamblador más incómodo para un ser humano, permite que las operaciones se denominen de manera más cómoda y reconocible para un programador (palabras como while, if, else...). Es decir, parte del lenguaje de programación tiene a tomar palabras de lenguajes como el inglés para facilitar el entendimiento del programador.

2. Construcción de traductores

Para programar, contamos con lenguajes de programación, como hemos visto en el punto anterior. El usar este lenguaje especial, que comparte similaridades con nuestro propio idioma, hace más sencillo y accesible el poder entenderlo y manejarlo. ¿Qué problema surge? El computador no es capaz de entender ese lenguaje directamente, ya que es más complejo, agrupa instrucciones...

Imagina que queremos hablar con un ruso, y le hablamos en inglés. Por mucho que para él sea más sencillo comprender el inglés que, por ejemplo, el español, sigue sin enterarse de nada. Claramente necesitamos un **traductor**, que conozca el inglés y sea capaz de pasarlo a ruso. Volviendo a un computador, necesitamos un traductor que transforme el lenguaje de alto nivel o de programación en lenguaje máquina (lenguaje que sí comprende el ordenador).

Ya que somos conscientes de la necesidad de un traductor, vamos a ver cómo trabaja y sus características con más profundidad.

Traductor

El **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. Reciben como entrada el lenguaje fuente, que define a una máquina virtual, y da como salida el lenguaje objeto, que define a una máquina real.

$$lenguaje.fuente --- TRADUCTOR -- > lenguaje.objeto$$

```
NOTA
Cuando hablamos lenguaje de alto nivel, estamos hablando de lenguaje que usa un
programador para programar; es más similar al lenguaje que usamos para
comunicarnos entre nosotros.

Cuando hablamos de lenguaje de bajo nivel nos referimos al que se acerca a lo que
entiende un ordenador, es decir, 0s y 1s.
```

Se han extendido dos formas de traducir lenguajes de programación, y por ellas distinguimos **dos tipos de traductores**:

1. **Compilador**

   Traduce todo el código de alto nivel a bajo nivel una vez, del tirón. Por tanto, genera una versión traducida de lo que hace el programa. Ahora bien, muchas veces es necesaria la interacción con el programador/usuario (pedir valores, confirmar actividades, seleccionar proceso a seguir...). Como ya se ha traducido el código primero, las entradas del usuario se reciben y traducen posteriormente, quedando al final como múltiples traducciones por separado.

   Necesitamos agrupar, enlazar, las traducciones que hemos hecho, y para resolver el problema surge el enlazador.

   - **Enlazador**

     Se encarga de ligar todas las traducciones que ha realizado el compilador, es decir, enlazar las instrucciones máquina. Al final generará un programa ejecutable para el computador, un archivo que el ordenador sí puede comprender.

2. **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.

   Genera una versión completa de la traducción teniendo en cuenta tanto el código como las entradas dadas por el usuario, todo a la vez. Traduce línea a línea en tiempo real; si tenemos una sentencia en la que necesita interacción del usuario, esperará a lo que de el usuario y lo añade a la traducción.

   No se almacena, por eso no se genera programa fuente. Simplemente estamos traduciendo en directo para usarlo en el momento.

   Quiero mencionar que más adelante hay un apartado exclusivo de los intérpretes, por lo que hay más información sobre ellos más adelante.

```
POSIBLES PREGUNTAS DE EXAMEN

*¿Cuándo usar un intérprete y cuando un compilador?
    El intérprete no genera programa objeto, mientras que el compilador sí, por
lo que si necesitamos ejecutar múltiples veces el código sin realizar en él
modificaciones va a ser óptimo el uso del  compilador. Por otro lado, si vamos a
necesitar interactuar mucho con el traductor (se pide mucha información al
usuario/programador) o vamos a necesitar hacer muchas modificaciones del código
el enlazador sería nuestra elección mejor.

*¿Cómo se unen los archivos que genera el compilador para lograr un único archivo
objeto?
    Claramente, el enlazador.

*¿Qué diferencia a un compilador de un intérprete?
    Que uno genera programa objeto (programa ejecutable) y el otro no, que uno
necesita agrupar diversos archivos con un enlazador y el otro no, que uno traduce
a la vez código y mensajes de usuario mientras que el otro lo hace en procesos
separados...
```

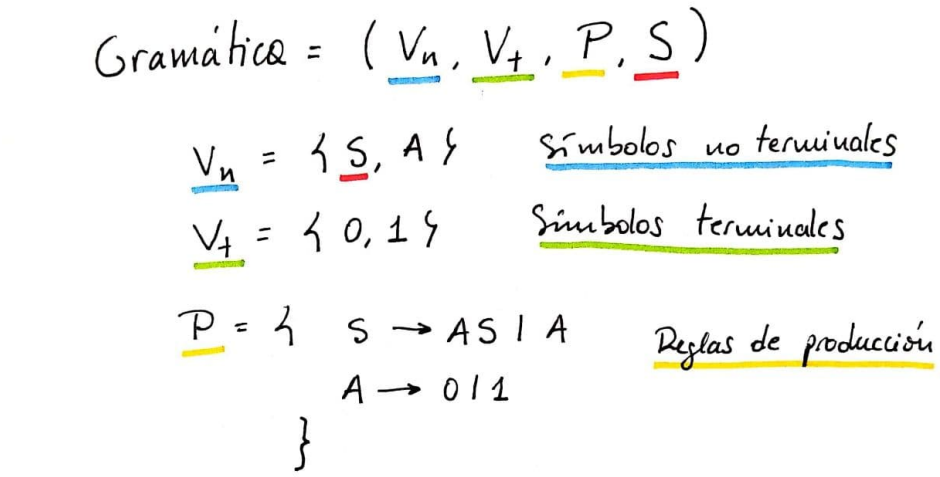
Gramáticas

La gramática, para cualquier idioma, es la estructura y leyes que organizan el uso de un lenguaje. Por ejemplo, en el español, algunas indicaciones que daría la gramática serían que los artículos preceden a los sustantivos, que los verbos tienen que estar conjugados de acuerdo al nombre que acompañan... Las gramáticas, aunque vengan algo precipitadas en esta parte del tema, son una herramienta importante para los traductores, como veremos más adelante.

Una **gramática** proporciona una especificación sintáctica precisa de un lenguaje de programación, es decir, las reglas que se deben seguir para programar con un lenguaje determinado y que tenga sentido. Dependiendo del tipo de gramática que tengamos aumentará la complejidad de la verificación sintáctica. Esto significa que cuanto más compleja la gramática, cuantas más reglas posea, más dificultosa se vuelve la sintaxis del lenguaje (el orden de los elementos en una oración, o en nuestro caso, en la orden).

La manera formal de describir gramáticas es mediante un grupo de elementos. Así pues, se definen las gramáticas como **G**= (Vn, Vt, P, S), donde cada parte determina:

- **Vn**: símbolos no terminales.
- **Vt**: símbolos terminales, los que solo pueden estar contenidos en la palabra
- **P**: representa al conjunto de reglas gramaticales que contiene la gramática, o reglas de producción. Estas indican cómo se pueden ir combinando los símbolos no terminales y terminales para poder obtener palabras o combinaciones válidas.
- **S**: axioma de la gramática, es decir, el símbolo inicial. Siempre debe estar contenido en Vn (los no terminales).



Para entendernos, es como si usáramos dos lenguajes (Vn y Vt), nos dijeran la primera letra (S) y nos dan una reglas (P) para desarrollar desde esa primera letra para llegar a lo que buscamos. La idea es que nosotros tenemos que ir añadiendo con las reglas de producción más letras a la palabra, hacer transformaciones, para quedarnos al final con una palabra que únicamente cuenta con símbolos terminales. Si una palabra tiene símbolos terminales y no se pueden transformar en otros significa que la gramática no la acepta.

Un detalle importante es cómo se escriben las reglas de producción. Siempre serán del estilo S->A o S::=A. Ambas significan lo mismo: podemos transformar la S en una A. Hay mil opciones, S::=0 transforma S en 0, A::=9S transforma A en 9S, Ax::=10S0 transforma Ax en 10S0... *IMPORTANTE* Hay que destacar que el profesor prefiere poner las reglas de producción con **::=** antes que con ->. En cambio, cuando más adelante escribamos todos las derivaciones que estamos haciendo las haremos con **->**: S -> AS -> 0S -> 0A -> 01.

Cuando se pone una regla como S::= A | 0S lo que estamos es resumiendo reglas. En el fondo son las reglas S::=A y S::=0S, pero al poner | podemos poner todas las opciones en las que se puede derivar S para escribir menor.

```
DEFINICIONES
    LETRA: cada símbolo que está en Vt o Vn. Puede ser un número, una letra, palabras clave del lenguaje de programación (if, else, for, while...) u otros elementos ("<expresion>", "palabra", "numero"...)

    PALABRA/CADENA: cada "cosa" que consigamos aplicando las reglas de producción (P) y siguiendo las reglas de la gramática. Una palabra de la gramática puede ser una linea de código, o un bucle for, una     secuencia if/else...

    DERIVAR: se denomina así a cada aplicación de una regla de producción sobre la palabra

    TERMINALES/NO TERMINALES: se llaman así porque si una palabra contiene al menos un símbolo no terminal no se considera que la palabra pertenezca a la gramática y por tanto no se acepta. De ahí el llamarlos terminales o no terminales

    ALFABETO: conjunto finito de símbolos
```

¿Cómo aplicamos la gramática para generar palabras del lenguaje?

1. Primero buscamos el axioma. Iniciamos desde ahí.
2. Debemos aplicar las reglas de producción de P en S. Aplicar las reglas se denomina derivar, y cuando se aplica se escribe como ->.
3. Aplicamos sucesivamente todas las reglas que queramos sobre la palabra, siempre que sea posible.
4. Paramos de aplicar reglas cuando lleguemos a obtener una palabra formada únicamente por símbolos terminales

Las palabras que se pueden conseguir partiendo de la S y aplicando las reglas de producción se consideran gramaticalmente correctas para esta gramática (son palabras que se aceptan). Si no conseguimos llegar a una palabra que no tenga símbolos no terminales no se considera de la gramática y por tanto no es una palabra correcta en el lenguaje (imagina en c++: si pones else {...} y después if (...){...} no se considera correcto; no es lo que la gramática de ese lenguaje establece como aceptable y dará error por no ser una expresión del lenguaje c++).

EJEMPLO

$$G = (\{0,1,2,3,4,5,6,7,8,9\}, \{N, C, A, B\}, N, P)$$

$$P = \{ N ::= 0 \mid C,$$
$$C ::= CA \mid A \mid CB,$$
$$A ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9,$$
$$B ::= 0 \}$$

En primer lugar vamos a identificar los elementos de la gramática.

1. **Símbolos no terminales**: está cambiado de orden en el ejemplo con respecto a lo explicado antes, pero si nos fijamos en las reglas de producción no habrá problema. Vemos que el conjunto que tienes las letras mayúsculas son las que, en las reglas de producción, se transforman en letras o números. Si sumamos el hecho de que el símbolo inicial es N (fíjate en cómo es el único símbolo que pertenece a un conjunto y también a la gramática), y que debe pertenecer al conjunto de no terminales (aunque también podría estar en la de los terminales, pero no es obligatorio), podemos afirmar que los símbolos no terminales de esta gramática son: {N, C, A, B}.
2. **Símbolos terminales**: el determinar los no terminales deja claro el resto, ya que los terminales serán el otro conjunto de elementos. En este caso, son los números: {0,1,2,3,4,5,6,7,8,9}.
3. **Símbolo inicial**: como hemos comentado antes, solo puede ser N, ya que está presente tanto en los símbolos no terminales como en los elementos de la gramática.
4. **Reglas de producción**: son las que identifica P, como se muestra en la foto.

Ahora que hemos identificado los elementos, debemos estudiar qué palabras acepta la gramática, para conocerla. Vamos a ir haciendo pasos de derivación uno por uno para ver qué es lo que podemos hacer:

- **N**: empezamos con el símbolo inicial. Aplicamos *N ::= 0*
- **0**: hemos llegado a un 0, la palabra solo contiene símbolos terminales por lo que hemos acabado

Sabemos que la gramática acepta al 0, pero con una sola palabra es complicado identificar qué permite la gramática; vamos a intentarlo de nuevo.

- **N**: de nuevo el símbolo inicial; aplicaremos ahora *N::=C*
- **C**: tenemos ahora una C, que según las reglas de derivación (o producción, valen ambos nombres) pueden ser varias opciones. Vamos a tomar *C::=CB*
- **CB**: vamos a intentar derivar sobre la C de nuevo, cogemos una nueva regla: *C::=CA*
- **CAB**: nos cansamos de generar Cs, así que la quitamos con la regla *C::=A*
- **AAB**: vemos que las A se pueden transformar en números entre 1 y 9, mientras que la B siempre es 0. Cambiemos la *B::=0*
- **AA0**: pues solo nos quedan As, vamos con la de en medio: *A::=7*
- **A70**: y acabamos con, por ejemplo, la regla *A::=2*
- **270**: solo símbolos terminales, ¡hemos acabado!

Este ejemplo fue uno de los realizados en clase, que define una gramática en la que podemos generar cualquier número natural (sin 0 a la izquierda, no significativos).

```
RECORDATORIO
    ::= significa que la parte de la izquierda puede reemplazarse por la parte de
la derecha

    | significa que puede transformarse a lo que pone a la izquierda de esta
señal o en lo de la derecha, es como el operador or
```

Para representar el proceso de derivación que realizamos podemos usar dos métodos diferentes, que son:

1. **Derivación**

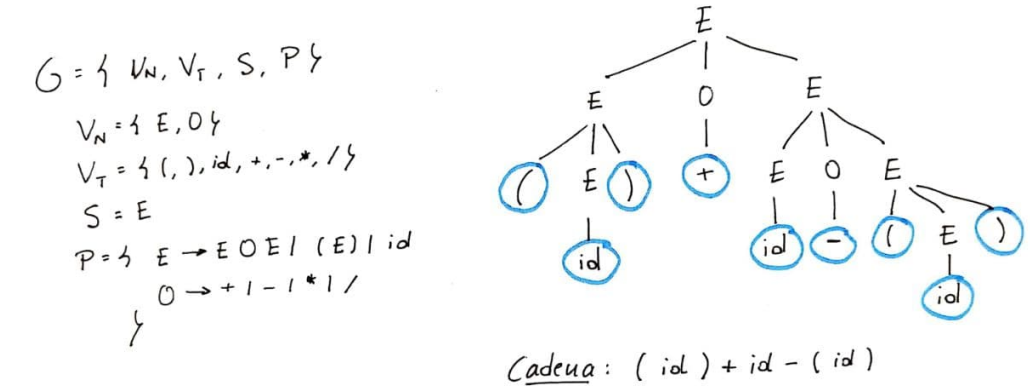
   En cada paso solo podemos aplicar una regla de producción.

   S -> AB -> AxB -> AxxB -> yxxB -> yxxz FIN

Vamos paso a paso, aunque podríamos obtener la misma cadena final aplicando las reglas en distinto orden.

2. **Árboles sintácticos**

   Se muestran como en el dibujo. Cada paso o decisión que se toma con las reglas que tenemos abren una rama nueva o varias de nuestro árbol, que crece hacia abajo. Es una opción más visual.



Características de las gramáticas

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.

Gramáticas ambiguas

Una gramática se denomina ambigua si, a partir de una producción dada, da lugar a más de un árbol sintáctico. No se puede determinar si una gramática es ambigua o no salvo si buscamos un ejemplo que concluya que hay ambiguedad.

¿Cuál es su problema? Que puede llevar a resultados no deseados, porque no se tiene en cuenta el orden que queremos. Imaginad un secuencia de cuentas como 5+2*3. Podría ejecutarse antes 5+2 y luego *3, y claramente el resultado es incorrecto. Eso implica que debemos modificar las reglas de producción para evitar que se den este tipo de problemas.

Por supuesto, no siempre tiene por qué ser un problema que una gramática sea ambigua. Si la gramática realiza sumas, por ejemplo, no importa el orden que tomemos ya que el resultado siempre será el deseado.

```
CUESTIONES
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?

Entiendo que se habla del orden en el que aparecen variables y operadores, ya que
dependiendo del caso podemos alterar el resultado. En ese caso es claro que el
uso de los paréntesis evita problemas con la solución.
```

Gramáticas libres de contexto

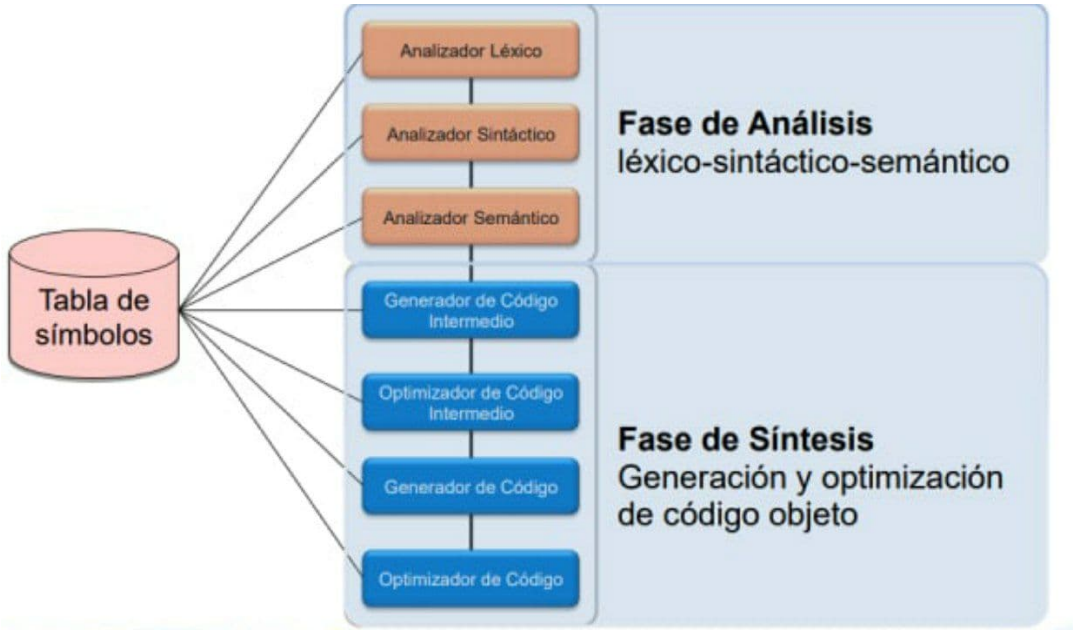
Son aquellas en las que las producciones o reglas de producción tienen una forma específica. Esto es, solo admite tener un símbolo no terminal en la parte izquierda de las producciones:

A ::= a A ::= aA

La denominación libre de contexto se debe a que donde aparezca A se podría poner a o aA, en este caso, independientemente del contexto en el que se encuentre A.

3. Proceso de compilación

El traductor no se va a poner a traducir directamente lo que le demos, ¡puede haber errores! La traducción no se hace de alto nivel a bajo nivel directamente, porque puede ser más complicado y llevar más tiempo, y porque hay veces en las que nuestro sistema necesita un paso intermedio. Necesitamos pues traducir de alto nivel a lenguaje intermedio, o de lenguaje intermedio a bajo nivel. Por eso contamos con dos fases de traducción: la **fase de análisis** y la **fase de síntesis**. Cada fase cuenta con una serie de analizadores, generadores u optimizadores.



La **fase de análisis**, como su nombre indica, está compuesto por tres analizadores de símbolos que se dividen en tres tipos: analizador léxico, sintáctico y semántico. La segunda fase, la **fase de síntesis**, cuenta con dos generadores y dos optimizadores: generador y optimizador de código intermedio y los generadores y optimizadores de código. A lo largo del apartado vamos a ver

cada uno de estos analizadores, generadores y optimizadores más extensamente para entender qué es lo que realmente están realizando.

La **tabla de símbolos** es la que mantiene la relación entre los distintos elementos (analizadores, generalizadores y codificadores). Todas las conclusiones que se sacan en cada parte se almacenan en la tabla de símbolos, por lo que la información que almacena es cada vez mayor (salvo que haya un error). Entre los analizadores, generadores... no se mandan información, son como funciones independientes sin relación entre ellas. Es la tabla de símbolos la que interactúa con ellos y almacena la información para que se pueda usar.

3.1 Análisis del Léxico

Su función principal es coger línea a línea el código, y en cada línea ir tomando palabra a palabra. Cada palabra la va clasificando en categorías, es decir, de **tokens**. Cada palabra clave tendrá una categoría, un token, y por supuesto, cada lenguaje tendrá un conjunto de tokens asociados. La definición formal de token sería "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)".

Por otra parte contamos con los **lexemas** o **palabras**, que son una secuencia de caracteres que identifican unívocamente a las palabras; cuentan con significado propio.

Por último tenemos al **patrón**, que es la estructura que define cada uno de los tokens. Es la descripción de la forma que pueden tomar los lexemas de un token.

El analizador léxico va clasificando las palabras por tokens y las introduce en la tabla de símbolos para que el analizador sintáctico las evalúe. Podemos contar con gran cantidad de tokens distintos: para palabras reservadas (if, else, while... un token para cada uno), para operadores (+, -, *...), para identificadores, constantes, paréntesis, corchetes...

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, ...`

El analizador busca un tipo de error concreto: elementos que no cuadren con los tokens permitidos. Cuando encuentre una palabra que no entre en ninguno de los patrones establecidos se avisará de error léxico. Un ejemplo típico es encontrar un carácter extraño en una palabra reservada, como whi¿le.

Si lo pensamos, podemos ver que el uso de gramáticas está presente aquí. La necesidad de establecer patrones para reconocer los tokens es clave, y claramente lo podemos llevar a cabo con una gramática. Generaríamos o usaríamos por tanto una por cada tipo de token.

Expresiones regulares

Para describir los patrones necesitamos algunas expresiones regulares que nos simplifiquen la tarea. Poner identificadores (empiezan por una letra y acepta después que hay letras o números indistintamente) se podría representar como: letra (letra | dígito)*. La * significaría que pueden ser tantas letras o dígitos como queramos, en el orden que queramos, pero siempre la palabra tiene que empezar con una letra. Por ver un par de ejemplos de token-lexema-patron, tenemos la siguiente tabla:

TOKEN	LEXEMA	PATRON
identificador	var21, myvar	letra(letra|patron)*
Operaciones-Mat	+,-,*,/	+|-|*|/
IF	if	"if"
PAR_DER	)	")"

3.2 Análisis Sintáctico

El análisis sintáctico busca, una vez tenemos los tokens, establecer si la estructura en la que se organizan los mismos es correcta o no. Para llevar a cabo su tarea hace uso de las gramáticas. Si nos fijamos, al final lo que creamos con las gramáticas son patrones aceptados por el lenguaje, y al igual que se pueden usar para verificar si las palabras (variables, operadores, palabras restringidas...) se pueden identificar con los tokens que tenemos, podemos usarlas para establecer si las líneas de código mantienen una estructura correcta.

Así pues, el principal objetivo del analizador sintáctico es analizar secuencias de tokens y comprobar si son correctas sintácticamente. Por cada secuencia de tokens que evalúa, el analizador nos dice si a secuencia es correcta o no sintácticamente (existe un conjunto de reglas gramaticales aplicables para poder estructurar la secuencia de tokens) y el orden en el que hay que aplicar las producciones de la gramática para obtener la secuencia de entrada.

Repasemos los dos hechos. En primer lugar, se avisa de si hay errores o de si todo era correcto. En caso de que no hubiese errores, el analizador procederá a dar a conocer el orden de las producciones, es decir, la secuencia de derivaciones que hay que realizar para obtener esa estructura de tokens. Concretamente, va a dar lugar un árbol sintáctico.

3.3 Análisis Semántico

Cuando hablamos de semántica de un lenguaje de programación estamos hablando del significado que le damos a las estructuras sintácticas que lo conforman. El significado será, por tanto, lo que representa cada sentencia del código.

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, ...). No solo tienen por qué mostrarse los problemas que vea el analizador como errores, sino que a veces echa mano de los warnings para avisar al programador de que puede estar ocurriendo algo que no desea.

¿Cuándo pueden ocurrir esos errores? Puede pasar por el valor de las variables (que sean tipo entero y le asignemos como valor un caracter, por ejemplo), o el alcance de las estas, es decir, si son locales o globales y estamos accediendo a ellas cuando no podemos o debemos. También puede ocurrir cuando se nos piden una serie de parámetros y damos menos.

3.4 Generación y optimización de código

En la generación de código se crea 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.

Una vez realizada la fase de generación se pasa a la optimización. 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.

El **optimizador** busca reducir en lo posible los elementos que causen un mayor tiempo de ejecución. Crear un optimizador no es algo sencillo, hay que tener muchas ideas y métodos para cada lenguaje.

Un ejemplo sencillo de optimización de código es modificar la posición de uan variable dentro del código: si estar declarada fuera de un bucle a estar dentro de él no afecta al correcto desarrollo del programa, sí que supone un cambio en el tiempo que se tarde en ejecutar dicho bucle. El ejemplo muestra a la izquierda el caso normal y a la derecha el optimizado, marcando entre * * la línea que modificamos:

```
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);
}
```

4. Intérpretes

Como vimos al principio del tema, un intérprete hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador.

Vimos antes algunas características destacables sobre los intérpretes, sobre todo focalizándolo en la búsqueda de diferencias con respecto al compilador. Para extender los conocimientos sobre los intérpretes, se muestran algunas características más sobre ellos:

- Como decíamos, no se genera archivo objeto almacenable en memoria para posteriores ejecuciones, sino que cada vez que queramos ejecutar el programa debemos pasar de nuevo por todos el proceso de análisis del intérprete
- Dado que no tenemos archivo objeto, la ejecución está supervisada en todos los casos por el intérprete
- Las instrucciones de los bucles se analizan tantas veces como indique el bucle, por lo que bucles muy extensos pueden suponer muchísimo tiempo de procesado para los intérpretes
- Solo es posible hacer optimizaciones a nivel de instrucción, no de estructuras, ni bloques, ni programas.

Retomemos la pregunta que hacíamos como posible de examen sobre cuándo usar un intérprete y cuando usar un compilador. Ahora que poseemos más conocimiento podremos tener una respuesta más extensa.

Cuándo usar un intérprete

1. El programador trabaja en un entorno interactivo y se desean obtener los resultados de la ejecución de una instrucción antes de ejecutar la siguiente
2. El programador lo ejecuta escasas ocasiones y el tiempo de ejecución no es importante.
3. Las instrucciones del lenguaje tiene una estructura simple y pueden ser analizadas fácilmente.
4. Cada instrucción será ejecutada una sola vez (por eso un gran ejemplo del uso de los intérpretes es Bash)

Cuándo usar un compilador

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

5. Modelos de Memoria de un proceso

Hemos hablado hasta ahora de cómo convertimos un archivo que hemos escrito con lenguaje de programación a uno ejecutable, conociendo la existencia de los traductores y todos los pasos que involucran su uso (analizadores, compiladores, enlazadores, optimizadores...). Damos un paso más y abordamos el uso de la memoria desde la generación del ejecutable hasta que se carga en memoria.

Nada se ha mencionado hasta ahora en el tema sobre el almacenado de variables, resultados... Pero el hecho es que no todas las variables son iguales, de manera que no será igual la manera de almacenar su información.

En el proceso de gestión de la memoria entran en juego muchos elementos: desde el lenguaje de programación que usemos, el compilador y el enlazador, el sistema operativo por supuesto, e incluso el hardware que mencionamos en otros temas que estaba hecho expresamente para la gestión de la memoria: el MMU.

Para que se unifique el manejo de la memoria y sea mucho más sencillo su uso se han establecido tres niveles destinados a su gestión: el nivel de **procesos**, el de **regiones** y el de **zonas**.

1. **Nivel de procesos**: se gestiona el reparto de memoria entre los procesos. Responsabilidad del SO
2. **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 compilado
3. **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.

Es claro que la memoria asociada a un proceso necesita cumplir muchas necesidades: tener un espacio lógico independiente, que exista la posibilidad de compartir memoria, que persistan los datos, que el espacio esté protegido... Una de las necesidades clave de la memoria de un proceso está relacionada con la **gestión del mapa de memoria de un proceso**, gestión que abarca desde

que se genera el ejecutable hasta su carga en memoria, y para ello debemos abordar más los niveles de regiones y zonas.

Tipos de datos

Como mencionábamos antes, los datos no son siempre iguales, de la misma manera que no se almacenan de manera similar. En memoria dividimos los datos en estáticos, dinámicos y heap.

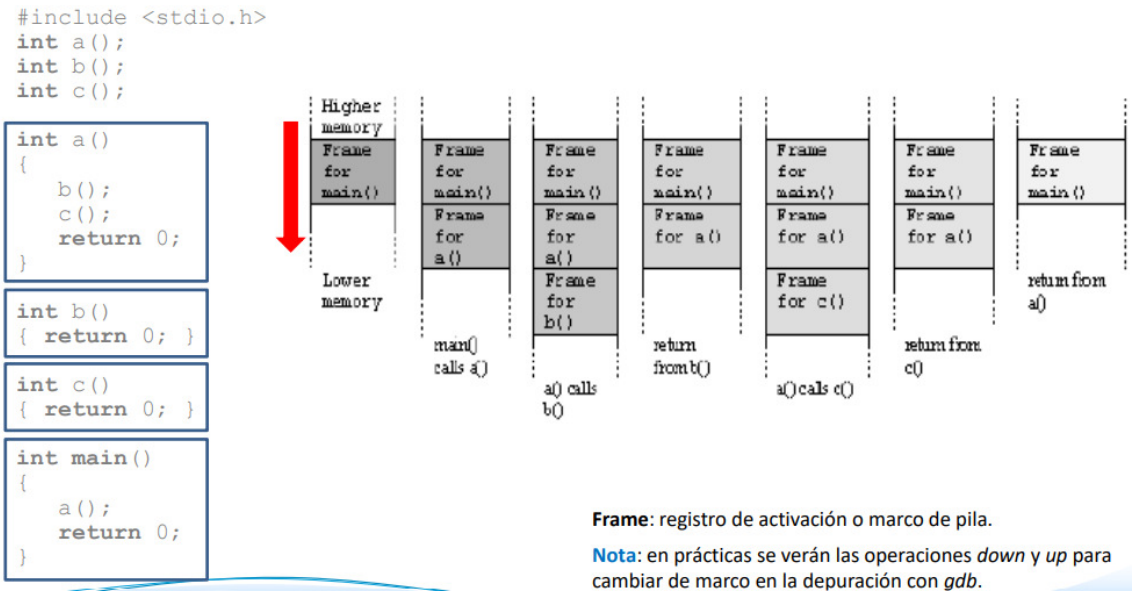
1. **Datos estáticos**: son globales a todo el programa, módulo o locales a una función (ámbito de visibilidad de una variable). Pueden ser constantes o variables, con o sin valor inicial. Implementación con direccionamiento absoluto o direccionamiento relativo (PIC – código independiente de la posición).
2. **Datos dinámicos**: 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. Están asociados a la ejecución de una función.
3. **Heap**: son datos dinámicos controlados por el programa. Es una zona de memoria usada en tiempo de ejecución que almacena todos los datos que no se dan (no se conocen aún) en tiempo de compilación (por ejemplo datos que el usuario da cuando se está ejecutando el programa), sino que se conocerán en tiempo de ejecución.

Tanto el heap como la pila son zonas de tamaño variable, y se van haciendo mayores o menores a lo largo de la ejecución, dependiendo de lo que necesiten.

Ejemplo de evolución de la pila

Vamos a ver un ejemplo de las diapositivas de clase, de cómo se modifica el tamaño de la pila

conforme realizamos la ejecución de un programa. En la imagen tenemos a la izquierda el código de cada una de las funciones que intervienen en el programa (a(), b(), c() y main()). A derecha encontramos cómo la memoria va añadiendo y quitando elementos. Debajo de la foto hay una breve explicación de lo que la imagen está intentando mostrar.



Se definen tres funciones, de manera que la a() llama a b() y c() para acabar devolviendo 0, b() devuelve 0 y c() también. El main llama a a(). Así pues se puede ver que estamos generando una cadena de llamadas a funciones del tipo main->a->b,c.

La memoria almacena el **frame** (registro de activación o marco de pila) de la función main. Cuando evalúa la función, se da cuenta de que necesita la función a, por lo que necesita añadir su frame. De igual manera ocurre con b. Cuando b termina, su registro de activación (o marco de pila) desaparece de memoria, es decir, ya no lo necesitamos. Luego se llama a c y se crea su registro de activación, se ejecuta, y se elimina su registro. Terminamos ejecutando a, para luego retirar su registro y finalmente realizar el main.

Código independiente de la posición (PIC)

Un fragmento de código es independiente de la posición 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.

Ejemplo de diferentes objetos de memoria

Hemos hablados hace nada de los tipos de datos en función de cómo se almacenan, peor lo más normal es que no haya quedado especialmente claro cómo identificar cada uno en un caso real. Veamos ahora entonces un caso real, y analicemos a partir del código qué variables serán estáticas y cuales dinámicas.

```
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  */
   }
   . . . .
}
```

Un dato importante es que **dentro del main todas las variables son estáticas**. Por esa razón todas las variables desde la a hasta la g son estáticas, independientemente del tipo de declaración que tengas, de si están inicializadas o de si son o no constantes. Como son estáticos se almacenan en la parte de datos estáticos. Importante recalcar que estático no es lo mismo que constante.

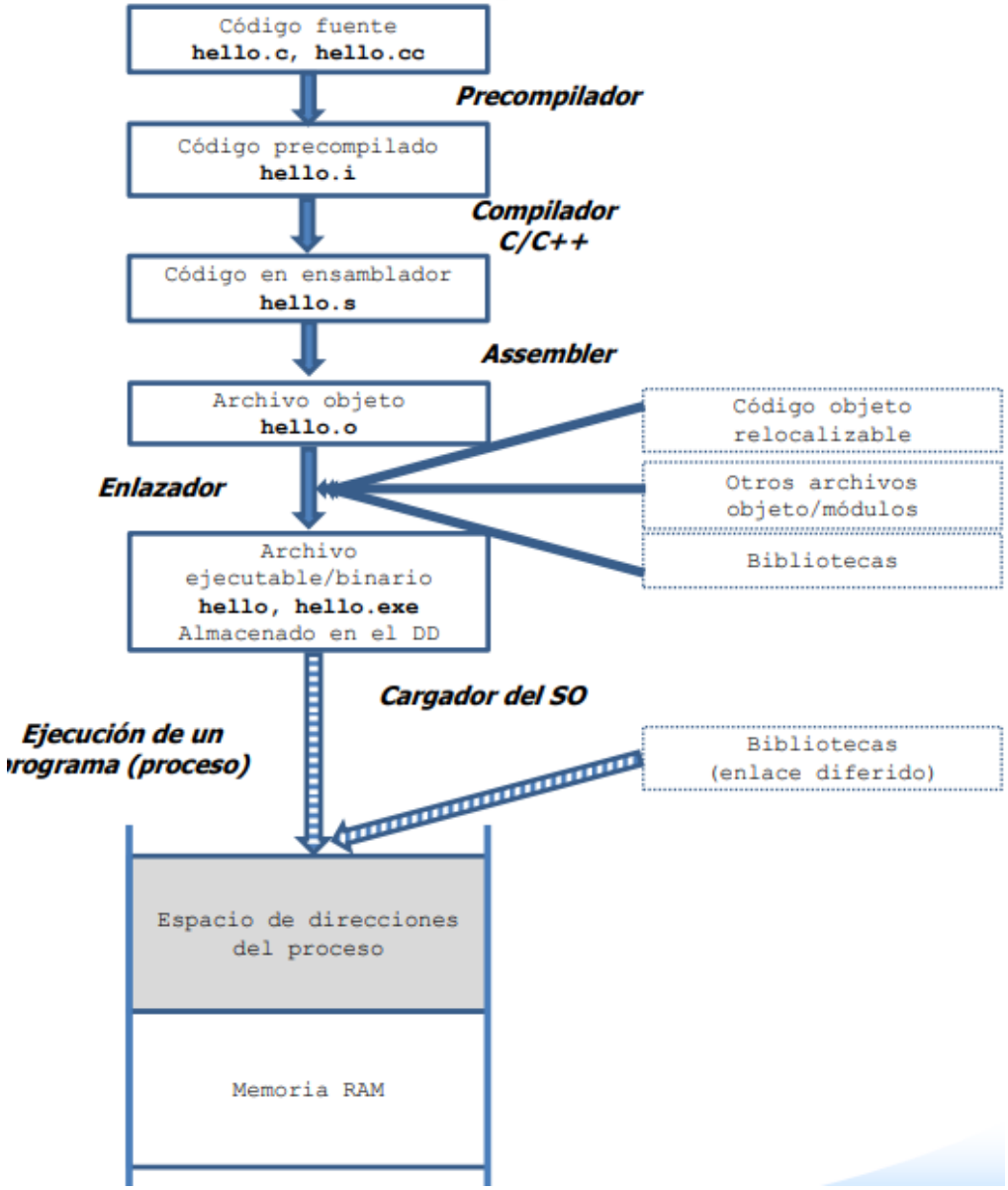
La variable h, por su parte, **depende de la función**, se denomina variable **dinámica de función**, por lo que se genera en la pila. Solo se usa cuando llaman a la función. De igual manera, las variables i y j solo se usan cuando se llaman a la función, por lo que son almacenadas en la pila como datos dinámicos.

Encontramos las variables k y l, que a pesar de estar **declaradas dentro de una función** se **definen estáticas**, por lo que aunque estén en la función se almacenan como datos estáticos. Por último, las variables m y n siguen siendo como i y j, por lo que son datos dinámicos.

6. Ciclo de vida de un programa

El ciclo de vida de un programa son todas las fases por las que tiene que pasar un programa hasta que se ejecuta. Básicamente, entender este ciclo es lo que motiva el temario de este tema. Mostrando en este punto el ciclo al completo se busca que, ya que hemos comprendido las fases más complejas o menos tratadas hasta el momento en la asignatura, tengamos una visión global del proceso y unifiquemos todo el conocimiento adquirido hasta ahora.

A fin de aclarar y simplificar el ciclo, se muestra a continuación un esquema vertical del proceso y una breve descripción de las fases.



A partir de un código fuente, un programa debe pasar por varias fases antes de poder ejecutarse, y algunas de ellas ya las hemos visto a lo largo del tema:

1. **Preprocesado**: transforma archivos .c a archivos .i en C
2. **Compilación**: traduce el código preprocesado en diferentes archivos que necesitan ser unidos, archivos .i a archivos .s en C
3. **Ensamblado**: unifica los archivos .s para generar un archivo objeto .o único
4. **Enlazado**: toma el archivo objeto .o, las bibliotecas .h, y si es necesario, añade código objeto relocalizable o módulos, todo ello para generar un archivo ejecutable en binario que ya puede interpretar el SO: genera los archivos .exe y a.out
5. **Carga y Ejecución**: el SO se encarga de realizar la ejecución de dicho programa cargándolo en memoria.

Ejemplo: cuando usamos un compilador como gcc, en verdad estamos usando un conjunto de programas que nos permiten realizar todos los pasos vistos, es decir, que nos permiten realizar el ciclo de vida y almacenar los datos intermedios (ficheros temporales, por ejemplo los .i o .s, que no tenemos por qué usar pero lo mismo nos son útiles para algo). Los comandos que hacen estas fases, según el ejemplo:

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
```

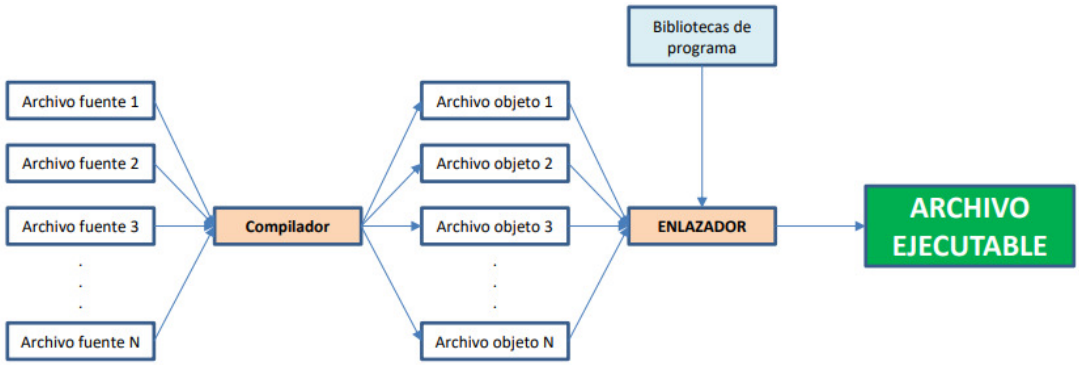
Compilación

El compilador se ha explicado y mencionado múltiples veces a lo largo del tema, pero vamos a extender un poco más la información que tenemos sobre él. Como sabemos, el compilador, traduce archivos de código fuente a código objeto. Las principales acciones que realiza son:

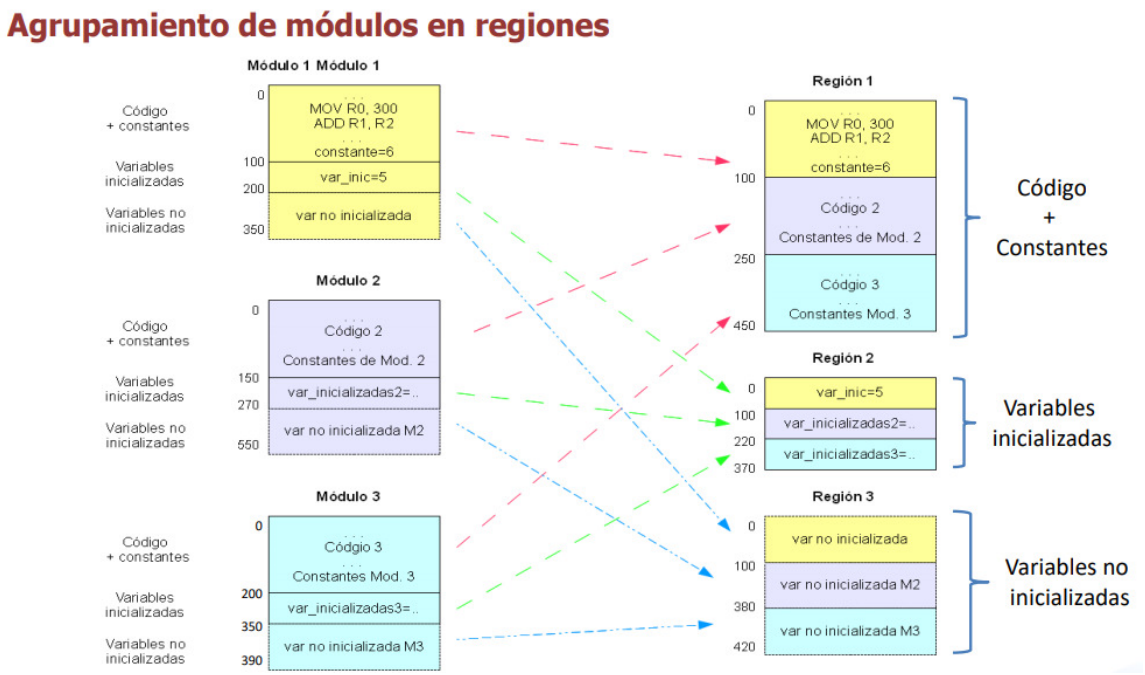
- Genera código objeto y calcula cuánto espacio ocupan los diferentes tipos de datos.
- Asigna direcciones a los símbolos estáticos (instrucciones o datos) y resuelve las referencias bien de forma absoluta o relativa (necesita reubicación)
- Las referencias a símbolos dinámicos se resuelven usando direccionamiento 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

Enlazado

Retomamos de nuevo el enlazador, dedo que el compilador y él van de la mano. Su función es agrupar los archivos objetos de la aplicación y las bibliotecas, y resolver las referencias entre ellos. No es algo tan sencillo como ponerlos uno detrás de otro, sino que se debe reordenar el código, creando una especie de código nuevo que unifique todo. Al final se unirán por trozos de código, por así decirlo. Por esta razón, en ocasiones debe realizar reubicaciones dependiendo del esquema de gestión de memoria utilizado.



Entre sus funciones se encuentra el completar la etapa de resolución de símbolos externos, agrupar las zonas de características similares para generar regiones...Une los ficheros de manera que el resultado tenga la forma de memoria como lo que veíamos de estático, dinámico, heap, pila... Además realiza la reubicación de módulos formando 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. Para que se vea mejor el reagrupamiento del que hablamos está la siguiente imagen:



Como puede verse, lo que se hace es tomar de cada módulo las diferentes partes y realizar nuevas agrupaciones que permitan el tener un único archivo que funcione como unidad y no dependa de nada más. Se puede apreciar cómo se está agrupando en la imagen por tipo de variable: inicializadas y no inicializadas. Además, también se unifican las constantes con el código.

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

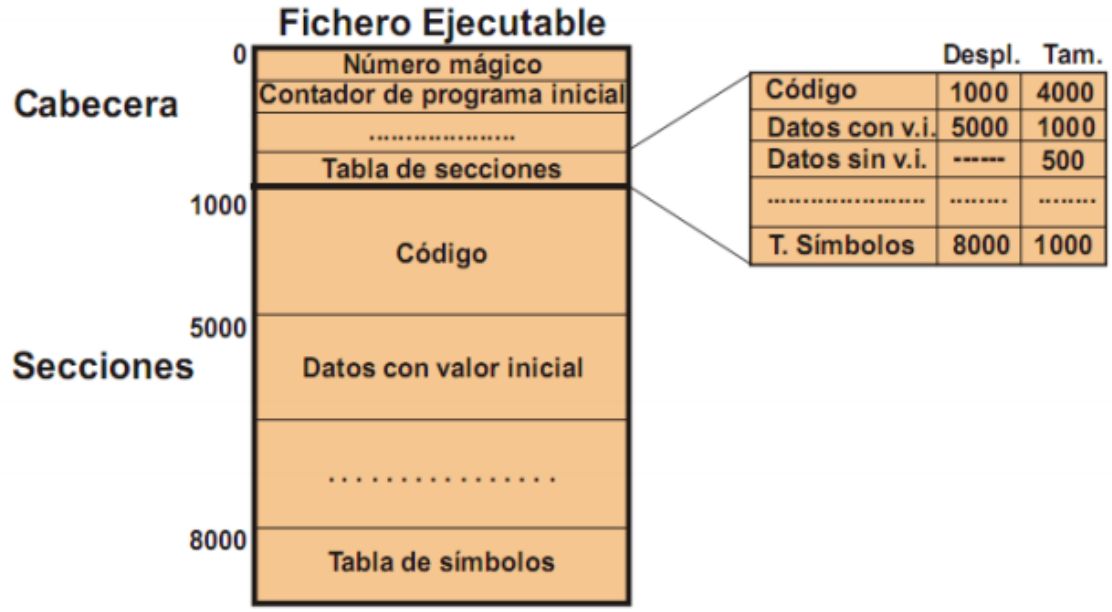
Carga en memoria principal y Ejecución

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** o **segmentación**.

Diferencias entre archivos objeto y ejecutable

Los archivos **objeto** son resultado de la compilación mientras que los **ejecutable** son producto del enlazado. Los archivos ejecutables cuentan en la cabecera el punto de inicio del mismo, es decir, la primera instrucción que se cargará en el PC, cosa que no ocurre en los archivos objeto. Además, solo en los ejecutables encontramos que en las regiones solo hay información de reubicación si ésta se ha de realizar en la carga.

Formato de un archivo ejecutable



Almacenar un archivo ejecutable puede tener su complicación debido a la cantidad de datos, variables, código... con los que contamos. Para estandarizar y simplificar el problema, dividimos el almacenado en dos grandes partes: **cabecera** y **secciones**.

La parte de cabecera cuenta con el **número mágico**: indica qué tipo de ejecutable es (ejecutable de microsoft, por ejemplo). Posee también el contador de programa inicial, e información que puede ser importante: cuantas variables tenemos almacenadas, si están o no inicializadas, el tamaño del código, el tamaño de la tabla de símbolos...

La parte de secciones contiene lo que realmente es el código y las variables, no solo información sobre ellas. Almacena el código completo, los distintos datos del programa y por último la tabla de símbolos.

Ejemplos de secciones de un archivo ejecutable ELF (readelf)

1. **.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.
2. **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
3. **.data – Variables globales y estáticas inicializadas**: Permisos: r y w
4. **.rdata o .rodata – Constantes o cadenas literales**
5. **.reloc – Información de reubicación para la carga**
6. **Tabla de símbolos**: Información necesaria (nombre y dirección) para localizar y reubicar definiciones y referencias simbólicas del programa. Cada entrada representa un símbolo

7. **Registros de reubicación**: Información utilizada por el enlazador para ajustar los contenidos de las secciones a reubicar.

7. Bibliotecas

Las bibliotecas son una colección de objetos, normalmente relacionados entre sí. Estas favorecen la modularidad y reusabilidad del código, facilitando el ciclo de vida del programa. Podemos clasificarlas en dos tipos.

Bibliotecas estáticas

Es básicamente un conjunto de archivos objeto que se copia en un único archivo que forma la biblioteca. Se ligan con el programa en la parte de enlazado (.a). Los pasos para su creación son:

1. Construimos el código fuente
2. Generamos el objeto
3. Archivamos el objeto (creamos la biblioteca)
4. Utilizamos la biblioteca

Tienen varios inconvenientes, entre los cuales el principal es el desperdicio de disco y memoria principal debido a que el código de la biblioteca está en todos los ejecutables que la usan. Por esta razón, los ejecutables que producen suelen ser de gran tamaño. Además tenemos que añadir que, dado que la información de la biblioteca estática se copia en cada archivo que la usa, cuando se actualizan o realizan modificaciones en la misma debemos recompilar los programas que la usan para evitar errores.

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 (se ligan con el programa en ejecución). Al ser así, los problemas de pérdida de espacio de memoria y de tener que recompilar el código, presentes en las estáticas, no aparecen en las dinámicas. Los pasos para su creación son:

1. Generamos el objeto de la biblioteca
2. Creamos la biblioteca
3. Usamos la biblioteca

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

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