Introducción a la construcción de compiladores

M. Luisa González Díaz*
Departamento de Informática
Universidad de Valladolid

1. Esquema general de un compilador

Los principios y técnicas que se usan en la escritura de compiladores se pueden emplear en muchas otras áreas. Se basan en los conceptos de teoría de autómatas y lenguajes formales que se están exponiendo en la parte teórica, y consituyen un campo de aplicación práctica bastante directo.

1.1. Objetivo de un compilador

El objetivo es, básicamente es *traducir* un programa (o texto) escrito en un lenguaje "*fuente*", que llamaremos *programa fuente*, en un equivalente en otro lenguaje denominado "*objeto*", al que llamaremos *programa* o *código objeto*. Si el programa fuente es correcto (pertenece al lenguaje formado por los programas correctos), podrá hacerse tal traducción; si no, se deseará obtener un mensaje de error (o varios) que permita determinar lo más claramente posible los orígenes de la incorrección.

La traducción podrá hacerse en dos formas:

interpretación: la traducción se hace "frase a frase"

compilación: la traducción se hace del texto completo

Concentraremos nuestra atención en el segundo de los métodos.

1.2. Entorno de un compilador

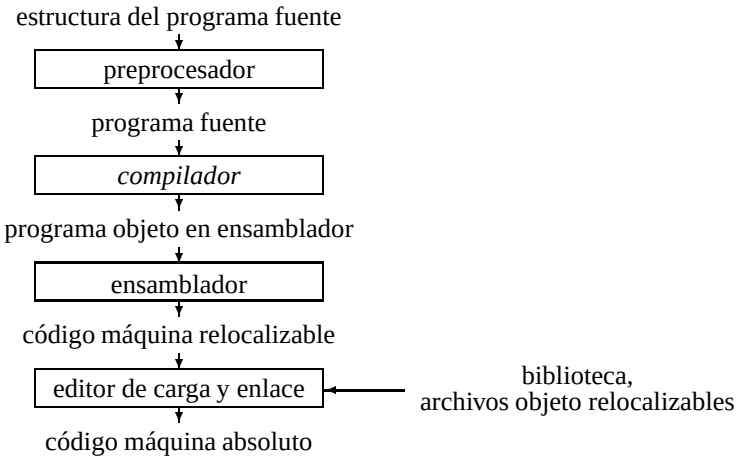
Es frecuente que, además del compilador, se utilicen otros programas para crear un código objeto ejecutable. Un esquema típico es el que se describe a continuación.

La estructura del programa fuente se escribe, usando algún programa de edición de texto (por ejemplo `vi`), y puede incluir texto en lenguaje fuente y algunas órdenes para el preprocesador. Este realizará algunas tareas, como eliminación de comentarios, expansión de macros (`#IF ...`), inclusión de archivos (`#include ...`), sustitución de constantes (`#define ...`), o algunas extensiones del lenguaje fuente.

El compilador traducirá el resultado del preproceso obteniendo un programa equivalente en lenguaje ensamblador, que a su vez será traducido por el ensamblador a *código máquina relocalizable*, en el cual las direcciones serán relativas a ciertas posiciones de origen, y quizás algunas llamadas a rutinas no estén resueltas.

Finalmente, el *editor de carga y enlace* (o "montador", o "link") resolverá las llamadas a rutinas, incluyéndolas a partir de otros objetos de biblioteca si procede, y obtendrá direcciones absolutas, de modo que ya se dispondrá del *código máquina absoluto* ejecutable.

*Esta sección es un resumen del capítulo I de "Compiladores: Principios, Técnicas y Herramientas" de Aho, Sethi y Ullman



No siempre todo este proceso es necesario, y, aún cuando se realice, no siempre será observado por el usuario, de forma que una sola orden (`cc programa.c`, por ejemplo) puede provocar el proceso completo. Los compiladores suelen incluir opciones que permiten obtener explícitamente (o conservar) los resultados intermedios.

Cada uno de los componentes es realmente un traductor. La dificultad máxima de traducción suele estar en la compilación propiamente dicha.

1.3. Estructura de un compilador

Cuando se ha hablado de programas *equivalentes* en distintos lenguajes (fuente y objeto), nos referimos a "tener el mismo significado" (aunque no fijemos ahora exactamente el concepto *significado*). Por ello, la compilación requiere dos grandes partes o tareas:

análisis : en la que se *analiza* el programa fuente para dividirlo en componentes y extraer de algún modo el significado

síntesis : en la que el significado obtenido se escribe en el lenguaje objeto

De las dos, la síntesis es la que requiere técnicas más especializadas. Durante el análisis, se determinan las operaciones que indica el programa fuente obteniendo una representación del significado, normalmente en una estructura jerárquica, de árbol, en la que cada nodo representa una operación, y cuyos hijos son los argumentos de dicha operación.

De este modo, a partir de la representación intermedia, diferentes partes de síntesis podrían obtener distintos códigos, para distintos lenguajes objeto; también, a partir de distintos lenguajes fuente, con partes de análisis adecuadas, se podría obtener una representación intermedia, que una parte de síntesis tradujera a su vez a un único lenguaje objeto. Nuevamente cada una de estas partes es también un traductor.

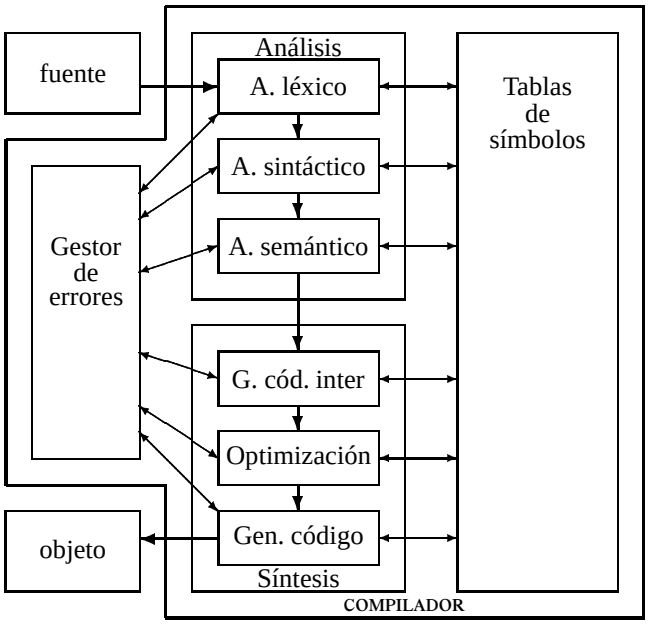
Aún se repite una vez más este esquema de traducciones intermedias en cada una de las partes: el análisis suele dividirse en tres *fases*: análisis léxico, análisis sintáctico y análisis semántico; y la síntesis en otras tres: generación de código intermedio, optimización de código y generación de código.

Se consideran por lo tanto 6 fases en el proceso de compilación, cada una de las cuales transforma la fuente de entrada, progresivamente, hasta conseguir el objeto final.

Por otra parte, cada una de las fases puede detectar errores, y debe informar adecuadamente de ellos. Consideraremos, por simplificar el esquema, un sólo bloque de manipulación de errores, que se usa desde cada fase, puesto que, además, la información de los errores deberá estar relacionada con el lugar del texto en el que se encuentra, relación que sólo puede establecer la fase de análisis léxico. La mayor parte de los errores se detecta en las dos primeras fases.

Además, las diferentes fases crean y acceden a una estructura de datos llamada *tabla de símbolos*, en la que se anotan a lo largo de las fases variadas informaciones o *atributos* que es necesario intercambiar; por ejemplo, las variables, con sus nombres, tipos, ámbito, posición relativa de memoria en la generación de código, etc.

Estos dos últimos elementos, aunque no sean realmente fases del proceso, tienen suficiente entidad como para ser consideradas bloques de similar importancia.



1.4. Fases de un compilador

Analizaremos algo más las fases usando como ejemplo simple el fragmento de código Pascal
`posicion := inicial + velocidad * 60`

1.4.1. Análisis léxico

La cadena de entrada se recibe como una sucesión de caracteres. El análisis léxico agrupa los caracteres en secuencias con significado colectivo y mínimo en el lenguaje, llamadas *componentes léxicos* (*palabras* o *"token"*), con ciertos atributos léxicos. En el ejemplo, se detectarían 7 :

1. el identificador `posicion`
2. el operador de asignación `:=`
3. el identificador `inicial`
4. el operador +
5. el identificador `velocidad`
6. el operador *
7. la constante numérica 60

Cada vez que se detecta un nuevo identificador, se anota una entrada en la tabla de símbolos.

El lenguaje de entrada tendrá un catálogo de componentes posibles, que se podrán codificar (por ejemplo mediante constantes enteras). Normalmente habrá al componentes como los siguientes:

- palabras reservadas: codificadas mediante las constantes BEGIN, END, IF, VAR ...
- identificadores: codificadas mediante la constante ID, y caracterizadas por ser sucesiones de letras y dígitos que comiencen por una letra, que no pertenezcan al catálogo de palabras reservadas. Se distinguirán unas de otras a partir de ahora por su posición en la tabla de símbolos (atributo léxico).
- operadores relacionales: codificados mediante la constante OPREL, que pueden ser uno de los siguientes: <=, <, >, >=, = <>. Para distinguirlos en fases posteriores se codificarán respectivamente con valores léxicos que serán otras constantes, que llamaremos MEI, MEN, MAY, MAI, IGU, DIF
- operadores aritméticos de un solo carácter: +, -, * .... Se codificarán con su propio código de carácter (lo que forzará a que el resto de las constantes deban usar números diferentes, pongamos superiores a 257)

La salida del analizador léxico es, entonces, una sucesión de pares: el tipo de componente léxico de que se trata y cuál de ellos es. Llamaremos componente léxico al primer elemento del par y valor léxico al segundo.

Por lo tanto, la salida del analizador léxico para el ejemplo sería:

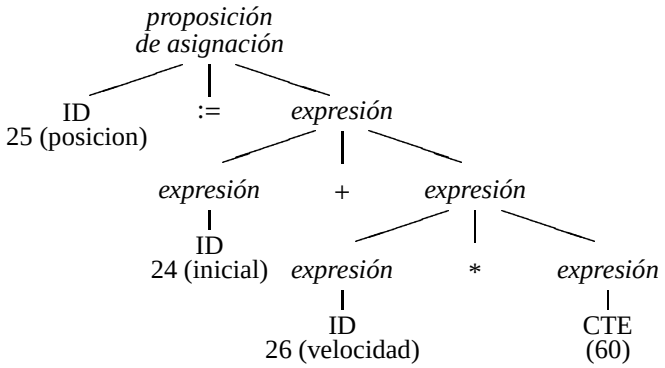
```
<ID, 25> <OPASIGN,> <ID, 24> <+,> <ID, 26> <*,> <CTE, 60>
```

mientras que la tabla de símbolos tendrá entradas para los tres identificadores (que realmente se habrán detectado antes, al leer la sección de declaración de variables, momento en el que se habrán instalado)

1	...	...
...		
24	inicial	...
25	posicion	...
26	velocidad	...
...		

1.4.2. Análisis sintáctico

Los componentes léxicos se agrupan para formar *frases*. El valor léxico de los componentes es en este momento irrelevante. Normalmente las frases se representan mediante una estructura de *árbol sintáctico*, siguiendo reglas que describen el lenguaje. Las reglas determinarán en el caso del ejemplo un árbol como el siguiente, en el que se muestra que la expresión es en primera instancia una suma y no un producto, dadas las precedencias de los operadores implicados.



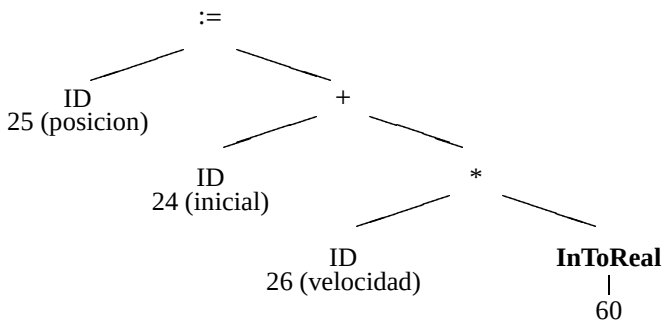
La división entre análisis léxico y sintáctico es algo arbitraria. Normalmente el criterio de división es la simplificación de ambas tareas, dejando las construcciones anidadas (paréntesis, operadores . . .) para el análisis sintáctico que utilizará técnicas algo más complejas que el léxico, que suele limitarse a construcciones de tipo regular.

1.4.3. Análisis semántico

En esta etapa se revisa el resultado del análisis sintáctico, recopilando por ejemplo información de tipos y construyendo una representación aún más abstracta. En el ejemplo que estamos considerando, en esta fase se anotará el tipo de los identificadores cuando se revise la declaración de variables (`var inicial, posision, velocidad: real`), de forma que la tabla de símbolos ahora contendrá más información:

1	. . .	. . .	. . .
. . .			
24	inicial	real	. . .
25	posicion	real	. . .
26	velocidad	real	. . .
. . .			
. . .			

Por otra parte, puesto que la expresión `60` corresponde a un entero, será necesario transformar su valor en real antes de realizar las operaciones (los algoritmos que operan con reales y enteros son distintos, dada la representación interna de los datos). La salida del analizador semántico puede ser entonces un *árbol de sintaxis abstracta*, en el que, más que la estructura sintáctica de la entrada, aparece la estructura de operaciones a realizar, como el siguiente:



1.4.4. Generación de código intermedio

Se genera en esta fase un código intermedio para una máquina abstracta, y es posible que explícitamente . Esta representación debe ser fácil de producir y fácil de traducir al programa objeto. Puede tener diversas formas. Una posible es la llamada *código de tres direcciones*, que consiste en una secuencia de intrucciones, cada una de las cuales involucra

- a lo sumo un operador (unario o binario), además de la asignación
- tres direcciones a lo sumo (las de los operandos y la del resultado)

Además, deberá generar nombres temporales para almacenar los resultados intermedios.

Para el ejemplo, la salida de esta fase podría ser:

temp1 := InToReal(60)
temp2 := id_{26} * temp1
temp3 := id_{24} + temp2
id_{25} := temp3

1.4.5. Optimación de código

Se trata en esta fase de *mejorar* el código, en el sentido de reducir la cantidad de resursos (tiempo y memoria) necesarios. Algunas optimaciones son triviales, como por ejemplo hacer algunas transformaciones directamente en la compilación, en lugar de dejarlo para la ejecución (sustituir `InToReal(60)` por 60.0). Otras pueden requerir un trabajo mucho mayor, pero mejorar significativamente la eficiencia, normalmente a costa de alejarse bastante del código original, como eliminar código inactivo (inaccesible), eliminar vaiables intermedias o resituar sentencias independientes de un bucle fuera de éste. Muchos compiladores permiten elegir la cantidad de optimación a realizar o no hacerla.

Para el ejemplo, una mejora sencilla lo convertiría en:

temp1 := id_{26} * 60.0
id_{25} := id_{24} + temp2

1.4.6. Generación de código

En esta fase final se genera por fin el código objeto, normalmente código máquina relocalizable o ensamblador. Se seleccionan entonces posiciones de memoria relativas o registros para las variables y cada sentencia del código intermedio se traduce a una secuencia de instrucciones que ejecutan la tarea. Por ejemplo:

```
MOVF 0068, R2
MULF #60.0, R2
MOVF 0060, R1
ADDF R2, R1
MOVF R1, 0064
```

suponiendo que en la asignación de posición relativa de las variables se hubieran asociado 60H, 64H y 68H a las variables que en la tabla de símbolos ocupan las entradas 24, 25 y 26 respectivamente (valores que se habrán anotado también en las casillas correspondientes de la tabla de símbolos).