COMPILADORES

Teoría e implementación

Jacinto Ruiz Catalán

COMPILADORES. Teoría e implementación
Jacinto Ruiz Catalán

ISBN: 978-84-937008-9-8

EAN: 9788493700898

RC Libros
Calle Mar Mediterráneo, 2
Parque Empresarial Inbisa, N-6 – P.I. Las Fronteras
28830 SAN FERNANDO DE HENARES, Madrid
Teléfono: +34 91 677 57 22
Fax: +34 91 677 57 22
Correo electrónico: info@rclibros.es
Internet: www.rclibros.es

Diseño de colección, preimpresión y cubierta: Grupo RC
Impresión y encuadernación: Gráficas Deva, S.L.
Depósito Legal: M-
Impreso en España

14-13 12 11 10 (03)

Prólogo

El presente libro pretende ser un manual de ayuda para estudiantes y estudiosos de procesadores de lenguajes y/o compiladores.

La teoría de compiladores es un mundo apasionante dentro de la informática. Pero a la vez complejo. El desarrollo de un compilador para un lenguaje medianamente potente es una tarea dura y costosa, tanto en tiempo como en recursos.

Pero al tiempo de ser una tarea costosa, puede ser muy gratificante, al implicar campos básicos de la informática como son la teoría de autómatas, de lenguajes, estructura y arquitectura de computadores, lenguajes de programación y algunos otros más.

Al construir un compilador, el desarrollador debe adentrarse en aspectos específicos de lo que es un computador y de lo que es un lenguaje de programación. Esto le da una visión profunda de aspectos clave del mundo de la informática.

Este libro pretende ocupar un espacio que está poco tratado, sobre todo en español. Se trata de la construcción de un compilador paso a paso, desde la especificación del lenguaje hasta la generación del código final (generalmente, un ejecutable). Hay muchos libros que tratan la teoría y algunos ejemplos más o menos complejos de compiladores. Existen también libros que desarrollan pequeños compiladores pero hasta ciertas fases, sin llegar a completar todo el proceso de desarrollo. Lo que pretendemos con este libro es dar las bases teóricas suficientes para poder abordar la construcción de un compilador completo, y luego implementarlo.

El libro consta de 5 partes, un prólogo y el índice.

En la parte I se analizan los aspectos teóricos de los procesadores de lenguajes y/o compiladores. Ocupa casi la mitad del libro. En estos ocho capítulos se desgranan las fases en las que se distribuye el proceso de creación de un compilador.

El capítulo 1 es una introducción, el capítulo 2 trata del análisis léxico, el 3 del sintáctico, el 4 del análisis sintáctico descendente, el 5 del ascendente, el 6 de la tabla de tipos y de símbolos, el 7 del análisis semántico y el 8 de la generación de código intermedio y final.

En las siguientes tres partes se desarrollan completamente sendos compiladores o traductores. Cada parte es más completa y compleja que la anterior.

En la parte II se desarrolla la creación de un traductor para un lenguaje de lógica de proposiciones. El lenguaje se llama **L-0**. En esta parte, el capítulo 9 trata la especificación del lenguaje, en el 10 se realiza el análisis léxico, en el 11 el análisis sintáctico y en el 12 el semántico y la generación de código.

En la parte III se desarrolla un compilador para un subconjunto de C. Le llamamos **C-0** y es bastante simple. Pero es necesario su estudio para poder abarcar la parte IV, en la que se construye un compilador para C más complejo (**C-1**). Se deja para el lector la construcción de un compilador aún más complejo, que podríamos llamarle **C-2**.

Dentro de la parte III, el capítulo 13 trata la especificación del lenguaje, el 14 el análisis léxico, sintáctico y semántico, el 15 la generación de código intermedio y el 16 la generación de código final.

La parte IV, en la que se desarrolla un compilador para C más complejo, **C-1**, consta del capítulo 17 en el que se trata la especificación del lenguaje, el capítulo 18 para el análisis léxico y sintáctico, el 19 para el análisis semántico y el 20 para la generación de código.

La parte V consta de dos apéndices y la bibliografía. El apéndice A explica las herramientas que se han utilizado en el desarrollo de los compiladores y el apéndice B explica las instrucciones de código intermedio y final en Ens2001 para el lenguaje **C-1**. Esta parte concluye con la bibliografía.

Este libro puede servir de material para un curso de un semestre sobre compiladores o para un curso de dos semestres. Si se utiliza para un semestre, que suele ser lo más normal para una Ingeniería Técnica en Informática, se puede estudiar la parte I (capítulos 1 a 7) y luego implementar el compilador **L-0** de la parte II (capítulos 9, 10, 11 y 12) y el compilador **C-0** de la parte III (sólo capítulos 13 y 14). Por último, el apéndice A.

El resto del libro se podría incluir en un curso de dos semestres, que suele ser lo habitual en un segundo ciclo de Ingeniería Informática.

- Curso de 1 semestre → Ingeniería Técnica en Informática → Capítulos 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14 y apéndice A.
- Curso de 2 semestres → Ingeniería Informática → Capítulos del 1 al 20 y apéndices A y B.

Espero que este libro sea de interés del lector y que pueda sacar provecho a su lectura. Tanto como he sacado yo al escribirlo.

Nota: El código fuente de los compiladores se encuentra en la página web: www.rclibros.es en la sección Zona de archivos, también lo puede solicitar al autor en su dirección de correo electrónico jacinruiz@hotmail.com.

Jacinto Ruiz Catalán

Ingeniero en Informática

Ingeniero Técnico en Informática de Sistemas

Baena, Octubre de 2009

PARTE I. TEORÍA

1. Introducción
2. Análisis léxico
3. Análisis sintáctico
4. Análisis sintáctico descendente
5. Análisis sintáctico ascendente
6. Tabla de tipos y de símbolos
7. Análisis semántico
8. Generación de código intermedio y final

CAPÍTULO 1

Introducción

1.1 Definición de compilador

Un compilador es un tipo especial de traductor en el que el lenguaje fuente es un lenguaje de alto nivel y el lenguaje objeto es de bajo nivel (figura 1.1).

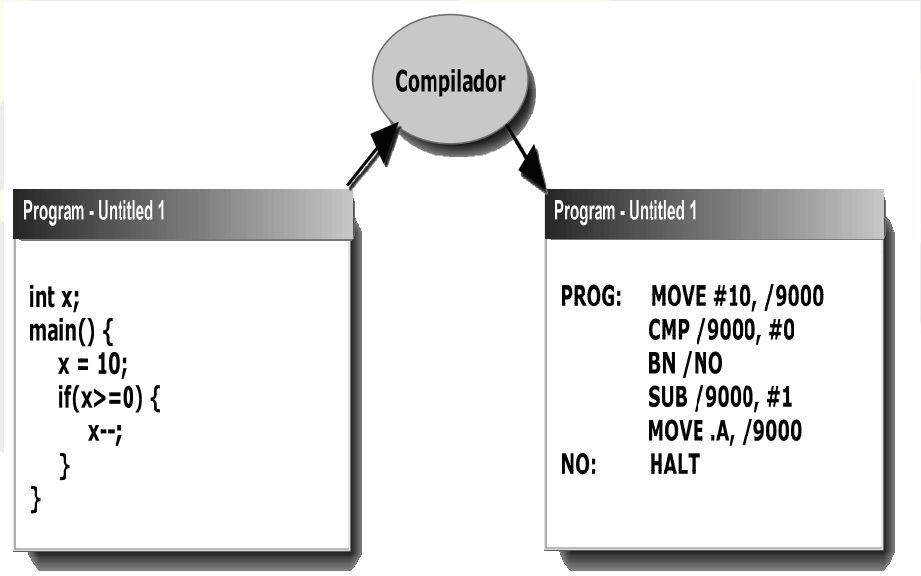

***Figura 1.1**. Esquema de un compilador*

Un traductor es un programa que convierte el texto escrito en un lenguaje en texto escrito en otro lenguaje (figura 1.2).

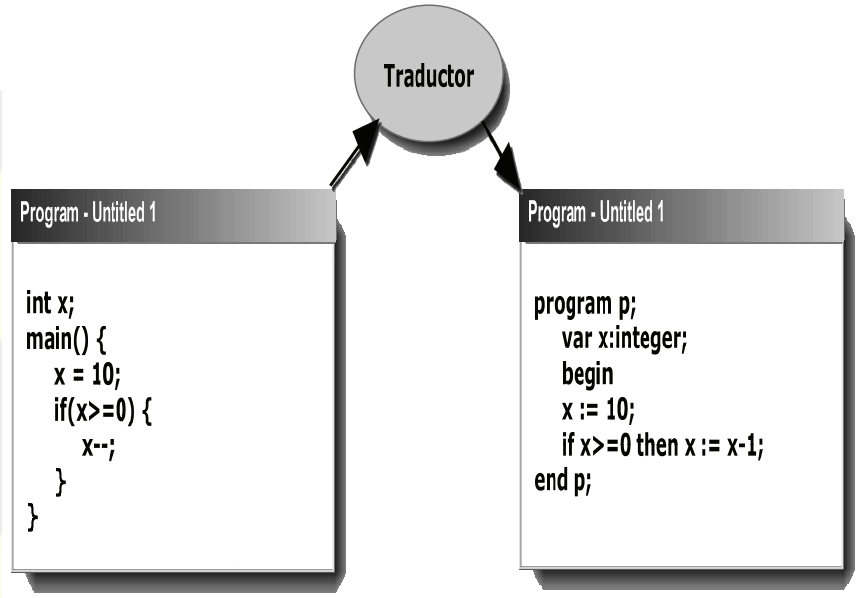

***Figura 1.2**. Esquema de un traductor*

Un ensamblador es un compilador donde el lenguaje fuente es un lenguaje ensamblador y el lenguaje objeto es el código de la máquina.

La diferencia entre compilador e intérprete es que el compilador analiza todo el programa fuente, crea el programa objeto y luego permite su ejecución (sólo del programa objeto obtenido) y el intérprete lee sentencia por sentencia el programa fuente, la convierte en código objeto y la ejecuta. Por lo tanto, es fácil comprender que tras compilar un programa, su ejecución es mucho más rápida que la ejecución de un programa interpretado.

Uno de los motivos de la existencia de programas interpretados es que hay algunos lenguajes de programación que permiten añadir sentencias durante la ejecución, cosa que no se podría hacer si fueran compilados. Algunos ejemplos son las versiones más antiguas de Basic y actualmente el lenguaje Phyton.

El compilador es asistido por otros programas para realizar su tarea, por ejemplo, se utiliza un preprocesador para añadir ficheros, ejecutar macros, eliminar comentarios, etcétera.

El enlazador se encarga de añadir al programa objeto obtenido, las partes de las librerías necesarias.

El depurador permite al programador ver paso a paso lo que ocurre durante la ejecución del programa.

Hay compiladores que no generan código máquina sino un programa en ensamblador, por lo que habrá que utilizar un programa ensamblador para generar el código máquina.

1.2 Estructura de un compilador

Un compilador es un programa complejo que consta de una serie de pasos, generalmente entrelazados, y que como resultado convierte un programa en un lenguaje de alto nivel en otro de bajo nivel (generalmente código máquina o lenguaje ensamblador).

Los pasos o fases de la compilación están actualmente bien definidos y en cierta medida sistematizados, aunque no están faltos de dificultad. Esta aumenta conforme se incrementa la riqueza del lenguaje a compilar.

Las fases del proceso de compilación son las siguientes (figura 1.3):

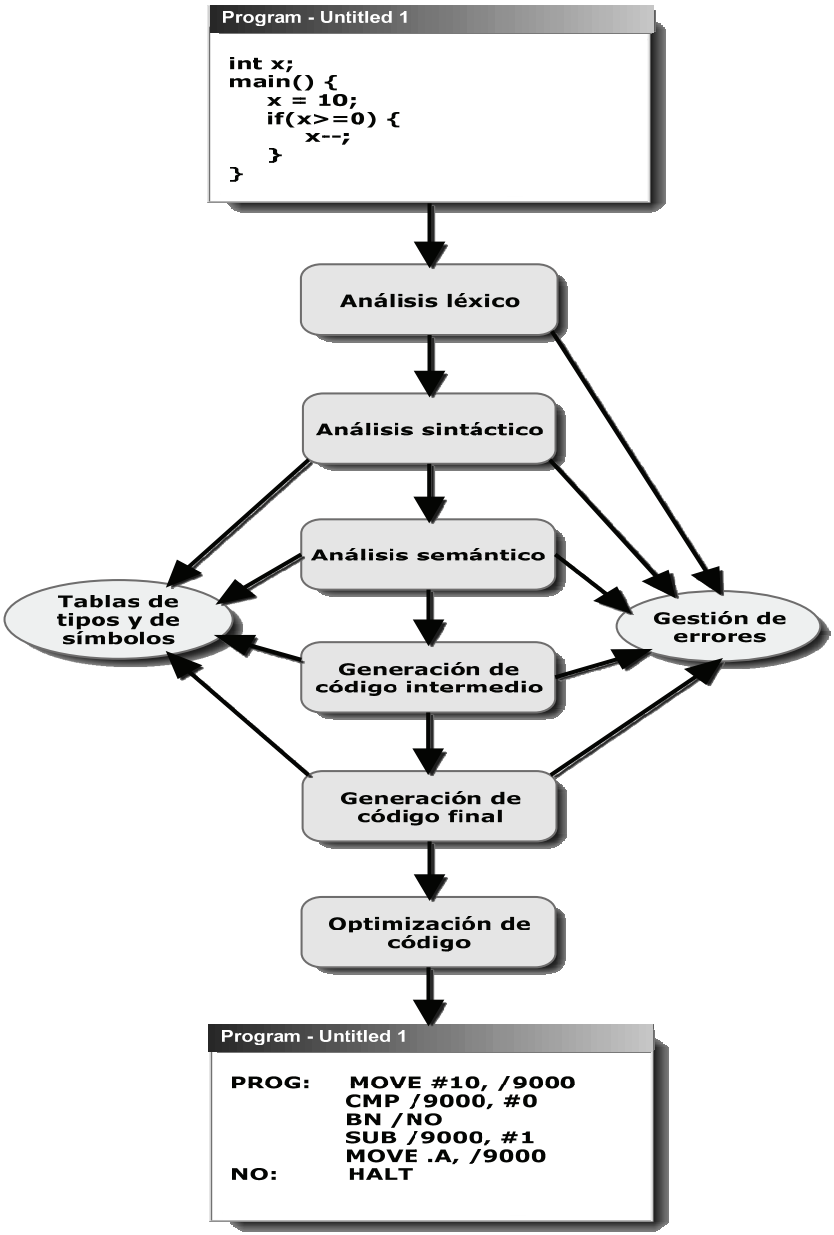

***Figura 1.3**. Fases del proceso de compilación*

1.2.1 Análisis léxico

Esta fase consiste en leer el texto del código fuente carácter a carácter e ir generando los *tokens* (caracteres relacionados entre sí). Estos *tokens* constituyen la entrada para el siguiente proceso de análisis (análisis sintáctico). El agrupamiento de caracteres en *tokens* depende del lenguaje que vayamos a compilar; es decir, un lenguaje generalmente agrupará caracteres en *tokens* diferentes de otro lenguaje.

Los *tokens* pueden ser de dos tipos; cadenas específicas como palabras reservadas, puntos y comas, etc., y no específicas, como identificadores, constantes y etiquetas. La diferencia entre ambos tipos de *tokens* radica en si ya son conocidos previamente o no. Por ejemplo, la palabra reservada *while* es conocida previamente en un lenguaje que la utilice, pero el nombre de una variable no es conocido anteriormente ya que es el programador el que le da nombre en cada programa.

Por lo tanto, y resumiendo, ***el analizador léxico lee los caracteres que componen el texto del programa fuente y suministra tokens al analizador sintáctico***.

El analizador léxico irá ignorando las partes no esenciales para la siguiente fase, como pueden ser los espacios en blanco, los comentarios, etc., es decir, realiza la función de preprocesador en cierta medida.

Los *tokens* se consideran entidades con dos partes: su tipo y su valor o lexema. En algunos casos, no hay tipo (como en las palabras reservadas). Esto quiere decir que por ejemplo, si tenemos una variable con nombre "x", su tipo es *identificador* y su lexema es *x*.

Por ejemplo, supongamos que tenemos que analizar un trozo de programa fuente escrito en lenguaje Java:

$x = x + y - 3;$

Los *tokens* suministrados al analizador sintáctico serían estos (el nombre que le demos a los tipos de *tokens* depende de nosotros):

- "x" : Tipo *identificador*, lexema *x*
- "=" : Lexema =
- "x" : Tipo *identificador*, lexema *x*
- "+" : Lexema +
- "y" : Tipo *identificador*, lexema *y*
- "-" : Lexema –
- "3" : Tipo *entero*, lexema *3*
- ";" : Lexema *;*

1.2.2 Análisis sintáctico

El analizador léxico tiene como entrada el código fuente en forma de una sucesión de caracteres. El analizador sintáctico tiene como entrada los lexemas que le suministra el analizador léxico y su función es comprobar que están ordenados de forma correcta (dependiendo del lenguaje que queramos procesar). Los dos analizadores suelen trabajar unidos e incluso el léxico suele ser una subrutina del sintáctico.

Al analizador sintáctico se le suele llamar *párser*. El *párser* genera de manera teórica un árbol sintáctico. Este árbol se puede ver como una estructura jerárquica que para su construcción utiliza reglas recursivas. La estructuración de este árbol hace posible diferenciar entre aplicar unos operadores antes de otros en la evaluación de expresiones. Es decir, si tenemos esta expresión en Java:

*x = x * y – 2;*

El valor de *x* dependerá de si aplicamos antes el operador producto que el operador suma. Una manera adecuada de saber qué operador aplicamos antes es elegir qué árbol sintáctico generar de los dos posibles.

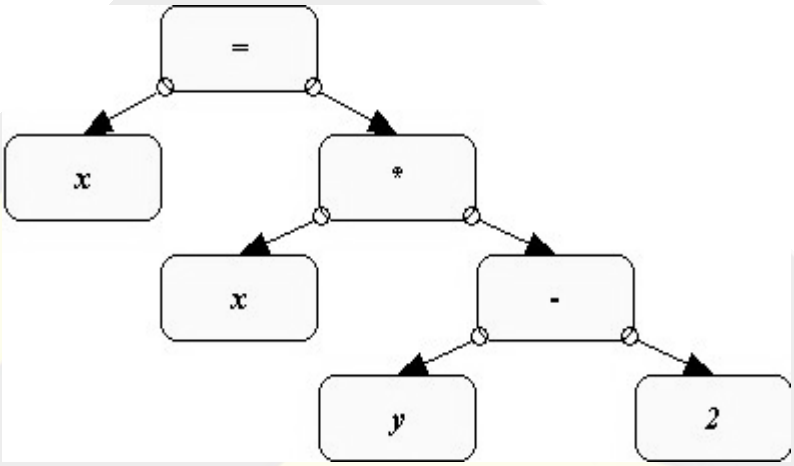

***Figura 1.4**. Arbol sintáctico*

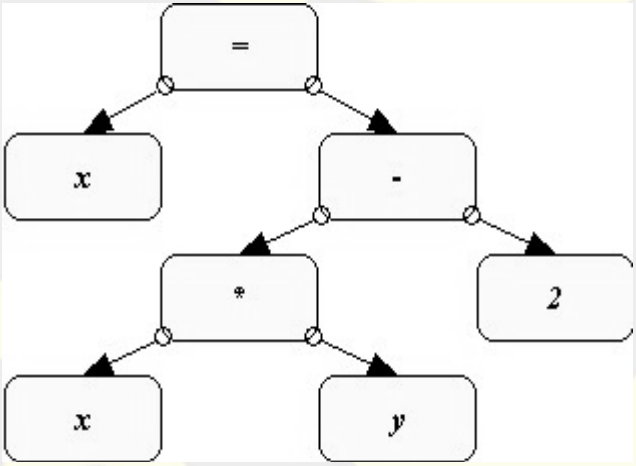

***Figura 1.5**. Arbol sintáctico*

En resumen, *la tarea del analizador sintáctico es procesar los lexemas que le suministra el analizador léxico, comprobar que están bien ordenados, y si no lo están, generar los informes de error correspondientes. Si la ordenación es correcta, se generará un árbol sintáctico teórico.*

1.2.3 Análisis semántico

Esta fase toma el árbol sintáctico teórico de la anterior fase y hace una serie de comprobaciones antes de obtener un árbol semántico teórico.

Esta fase es quizás la más compleja. Hay que revisar que los operadores trabajan sobre tipos compatibles, si los operadores obtienen como resultado elementos con tipos adecuados, si las llamadas a subprogramas tienen los parámetros adecuados tanto en número como en tipo, etc.

Esta fase debe preparar el terreno para atajar las fases de generación de código y debe lanzar los mensajes de error que encuentre. En resumen, su tarea es revisar el significado de lo que se va leyendo para ver si tiene sentido.

Esta fase, las anteriores y las siguientes se detallarán más adelante, en el desarrollo de los otros capítulos.

1.2.4 Generación de código intermedio

El código intermedio (CI) es la representación en un lenguaje sencillo (incluso inventado por el creador del compilador) de la secuencia real de las operaciones que se harán como resultado de las fases anteriores.

Se trata de representar de una manera formalizada las operaciones a llevar a cabo en un lenguaje más cercano a la máquina final, aunque no a una máquina concreta sino a una máquina abstracta. Es decir, no consiste en generar código ensamblador para una máquina basada en un procesador M68000, por ejemplo, sino en generar código que podría luego implementarse en cualquier máquina. Este lenguaje intermedio debe de ser lo más sencillo posible y, a la vez, lo más parecido a la manera de funcionar de la máquina final.

Hay dos ventajas clave por las que se debe utilizar la generación de código intermedio:

- Permite crear compiladores para diferentes máquinas con bastante menos esfuerzo que realizar todo el proceso para cada una de ellas.
- Permite crear compiladores de diferentes lenguajes para una misma máquina sin tener que generar cada vez el código final (puesto que tenemos ya el código intermedio creado).

Se suele utilizar una forma normalizada de instrucciones para este lenguaje intermedio que se llama *código de tres direcciones*.

Este código tiene su origen en la necesidad de evaluar expresiones, aunque se utiliza para todo el proceso (modificando el significado inicial de dirección).

Veamos un ejemplo aclaratorio. Supongamos que tenemos una máquina teórica que consta de registros numerados del R1 en adelante que pueden contener valores de tipo entero. Pensemos que tenemos este código en Java:

x = x +1;

Sea que *x* representa un valor entero y que hay un método que nos convierte el lexema de *x* en su valor entero. Supongamos que 1 es un lexema que representa otro valor entero, en este caso el valor 1 y que tenemos un método para convertir el lexema en su valor entero. Supongamos también que tenemos un método para convertir un número entero en su lexema.

Los pasos para generar el CI de la sentencia anterior serían:

- R1 = valor(x)
- R2 = valor(1)
- R3 = R1 + R2
- x = lexema(R3)

Estas operaciones las podríamos representar con el siguiente código intermedio:

- CARGAR x null R1
- CARGAR 1 null R2
- SUMAR R1 R2 R3
- CARGAR R3 null x

Si revisamos lo que hemos hecho, veremos que hay dos tipos de instrucciones de CI, una para cargar y otra para sumar (en el capítulo en que explicamos más detalladamente la generación de CI veremos que hay bastantes más instrucciones). Pero las dos tienen un punto en común, constan de un identificador de instrucción y de tres parámetros, aunque alguno de ellos puede ser nulo.

Cualquier expresión puede ser representada por una o varias de estas instrucciones de tres direcciones. Las llamamos de tres direcciones porque en

realidad se utilizan direcciones de memoria (llamadas temporales) y no registros.

El código de tres direcciones consta de un identificador de código, dos direcciones de operandos y una dirección de resultado de la operación.

Para realizar cualquier operación, es suficiente con generar diferentes instrucciones de código de este tipo, por lo que podemos tener un CI sencillo, versátil y útil.

Veremos un caso real para ver cómo funciona:

Supongamos que tenemos un programa en lenguaje C que consiste sólo en declarar la variable *x*. Cargarla con el valor 0 y luego sumarle 1. Supongamos que nuestro programa se va a alojar a partir de la dirección de memoria 0. Vamos a guardar los contenidos de las variables a partir de la dirección 9000 y vamos a utilizar las direcciones a partir de la 10000 como direcciones temporales.

El programa sería algo así:

```
int x
main() {
        x = 0;
        x = x + 1;
}
```

Se podría generar este código intermedio:

```
CARGAR 0 null 10000
CARGAR 10000 null 9000
CARGAR 9000 null 10001
CARGAR 1 null 10002
SUMAR 10001 10002 10003
CARGAR 10003 null 9000
```

Podemos ver que vamos utilizando direcciones temporales conforme las vamos necesitando para guardar resultados parciales.

Al final de todo, vemos que hemos recuperado el resultado parcial guardado en la dirección 10003 y lo hemos cargado en la dirección donde guardábamos el valor de la variable *x*. Por lo que al final, el valor de *x* es 1, que es el contenido de la dirección 9000.

La optimización de código intermedio consiste en el proceso de ir reutilizando estas direcciones temporales para que el consumo de memoria no se dispare. Además, consiste en optimizar el código generado para reducir el número de instrucciones necesarias para realizar las mismas operaciones.

1.2.5 Generación de código final

La generación de código final (CF) es un proceso más mecánico, ya que consiste en ir pasando las distintas instrucciones de CI (que suelen ser de pocos tipos diferentes) al lenguaje ensamblador de la máquina que se vaya a utilizar (más adelante, se puede ensamblar el código y obtener un ejecutable, pero este proceso ya es automático).

Dependiendo de la cantidad de memoria disponible o del número de registros disponibles, quizás sea necesario utilizar en vez de direcciones de memoria como temporales, registros, ya que así se reduce la memoria a usar. Esto es recomendable sobre todo para programas grandes porque casi todas las operaciones que se realizarán van a ser de cargas y almacenamientos de valores.

1.2.6 Tablas de símbolos y de tipos

Ponemos aquí esta sección, aunque no es una fase del proceso de compilación, porque es una parte importantísima de todo el proceso.

La tabla de símbolos es una estructura de datos auxiliar donde se va a guardar información sobre las variables declaradas, las funciones y procedimientos, sus nombres, sus parámetros y en generar cuanta información vaya a ser necesaria para realizar todo el proceso de compilación.

La tabla de tipos no es menos importante, ya que va a guardar información tanto sobre los tipos básicos como sobre los tipos definidos en el programa a compilar.

El compilador debe tener acceso a estas tablas a lo largo de todo el proceso de compilación. Además, el número de accesos suele ser alto, por lo que es conveniente optimizar la estructura de estas tablas para que su manipulación sea lo menos costosa en tiempo y así reducir el tiempo de compilación (aunque no habrá variación en el tiempo de ejecución del programa compilado ya que el proceso de manipulación de las tablas de tipos y símbolos se hace en tiempo de compilación).

1.2.7 Manejo de errores

El manejo de errores es vital para el proceso ya que informa al programador dónde hay errores, de qué tipo son, etc. Cuanta más información sea capaz de dar al programador, mejor.

Por lo tanto, no hay que dejar atrás este aspecto del proceso de compilación.

La detección de errores no siempre es posible en la fase en que se producen, algunas veces se detectarán en fases posteriores y algunos de ellos no se podrán detectar ya que se producirán en el tiempo de ejecución (serán responsabilidad del programador que ha realizado el código fuente).

Generalmente, es en las fases de análisis sintáctico y semántico donde suelen aparecer y *es importante que el proceso no se pare al encontrar un error,* sino que continúe su proceso de análisis hasta el final y luego informe de todos los errores encontrados.

1.3 Fases del proceso de compilación

Aunque las fases del proceso son las descritas en el epígrafe anterior, estas pueden agruparse desde un punto de vista diferente. Se pueden agrupar en una fase inicial o *front-end*, y en una fase final o *back-end*. El *front-end* agrupa las fases dependientes del lenguaje fuente e independientes de la máquina final y el *back-end* las fases independientes del lenguaje fuente pero dependientes del código intermedio y de la máquina de destino.

Cada fase comienza con un código y termina con otro diferente. El *front-end* comienza con el código fuente y finaliza con el código intermedio. Y el *back-end* comienza con el código intermedio y acaba con el código final.

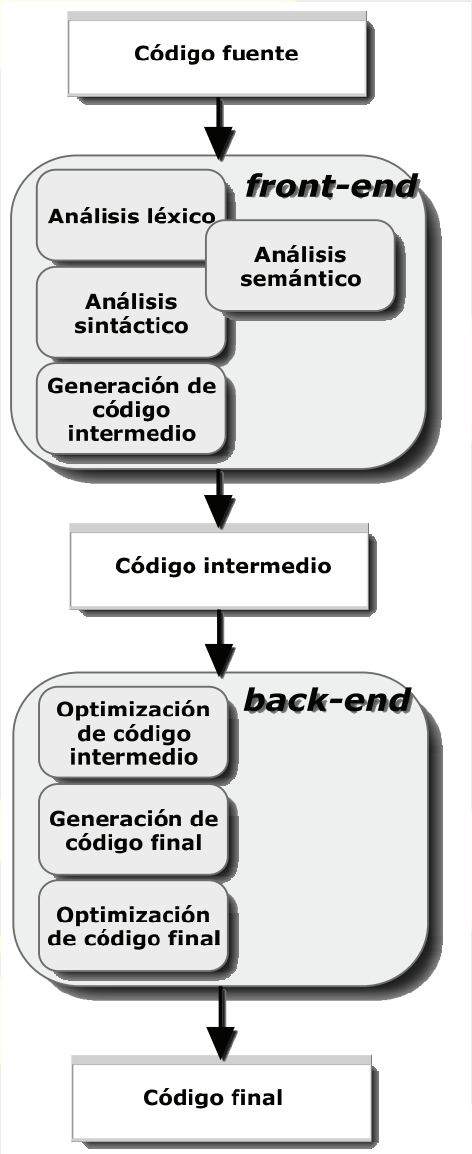

***Figura 1.6**. Front-end y back-end*

1.4 Herramientas y descripción del lenguaje

A la hora de hacer todo el proceso de confección de un compilador se suelen utilizar herramientas, que al no abarcar todas las fases, suelen combinarse. Aparte de las herramientas, se debe utilizar un lenguaje base para programar los pasos a seguir. Este lenguaje base debe ser compatible con las herramientas que utilicemos.

Por otra parte, antes de nada hay que saber para qué lenguaje vamos a hacer el compilador. Debemos definir su especificación léxica (los lexemas que utiliza), su especificación sintáctica (la ordenación válida de esos lexemas) y su especificación semántica (descripción del significado de cada lexema y las reglas que deben cumplirse).

También debemos saber si existe ya un código intermedio que podamos utilizar o hay que crear uno nuevo (este aspecto se suele hacer cuando llegamos a la fase correspondiente) y el código final que utilizaremos (también se suele hacer durante la fase de generación de código intermedio y generación de código final).

Otro aspecto que debemos tener en cuenta es el lenguaje base que vamos a utilizar, que dependerá tanto de nuestras preferencias como de las herramientas de ayuda que utilicemos.