Capítulo 1

Introducción

1.1 Visión general

Uno de los principales mecanismos de comunicación entre un ordenador y una persona viene dado por el envío y recepción de mensajes de tipo textual: el usuario escribe una orden mediante el teclado, y el ordenador la ejecuta devolviendo como resultado un mensaje informativo sobre las acciones llevadas a cabo.

Aunque la evolución de los ordenadores se encuentra dirigida actualmente hacia el empleo de novedosas y ergonómicas interfaces de usuario (como el ratón, las pantallas táctiles, las tabletas gráficas, etc.), podemos decir que casi todas las acciones que el usuario realiza sobre estas interfaces se traducen antes o después a secuencias de comandos que son ejecutadas como si hubieran sido introducidas por teclado. Por otro lado, y desde el punto de vista del profesional de la Informática, el trabajo que éste realiza sobre el ordenador se encuentra plagado de situaciones en las que se produce una comunicación textual directa con la máquina: utilización de un intérprete de comandos (*shell*), construcción de ficheros de trabajo por lotes, programación mediante diversos lenguajes, etc. Incluso los procesadores de texto como WordPerfect y MS Word almacenan los documentos escritos por el usuario mediante una codificación textual estructurada que, cada vez que se abre el documento, es reconocida, recorrida y presentada en pantalla.

Por todo esto, ningún informático que se precie puede esquivar la indudable necesidad de conocer los entresijos de la herramienta que utiliza durante su trabajo diario y sobre la que descansa la interacción hombre-máquina: el **traductor**.

Existe una gran cantidad de situaciones en las que puede ser muy útil conocer cómo funcionan las distintas partes de un compilador, especialmente aquélla que se encarga de trocear los textos fuentes y convertirlos en frases sintácticamente válidas. Por ejemplo, una situación de aparente complejidad puede presentársenos si se posee un documento de MS Word que procede de una fusión con una base de datos y se quiere, a partir de él, obtener la B.D. original. ¿Cómo solucionar el problema? Pues basándose en que la estructura del documento está formada por bloques que se repiten; la solución podría ser:

- Convertir el documento a formato texto puro.
- Procesar dicho texto con un traductor para eliminar los caracteres superfluos y dar como resultado otro texto en el que cada campo de la tabla de la B.D. está entre comillas.
- El texto anterior se importa con cualquier SGBD.

Otras aplicaciones de la construcción de traductores pueden ser la creación de preprocesadores para lenguajes que no lo tienen (por ejemplo, para trabajar fácilmente con SQL en C, se puede hacer un preprocesador para introducir SQL inmerso), o incluso la conversión del carácter ASCII 10 (LF) en "
" de HTML para pasar texto a la web.

En este capítulo, se introduce la construcción de un compilador y se describen sus componentes, el entorno en el que estos trabajan y algunas herramientas de software que facilitan su construcción.

1.2 Concepto de traductor.

Un traductor se define como **un programa que traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa equivalente escrito en un lenguaje destino produciendo, si cabe, mensajes de error**. Los traductores engloban tanto a los compiladores (en los que el lenguaje destino suele ser código máquina) como a los intérpretes (en los que el lenguaje destino está constituido por las acciones atómicas que puede ejecutar el intérprete). La figura 1.1 muestra el esquema básico que compone a un compilador/intérprete.

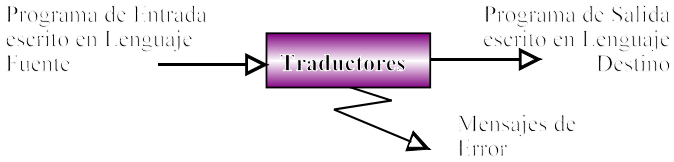

Figure 1 Esquema preliminar de un traductor

Es importante destacar la velocidad con la que hoy en día se puede construir un compilador. En la década de 1950, se consideró a los traductores como programas notablemente difíciles de escribir. El primer compilador de Fortran (*Formula Translator*), por ejemplo, necesitó para su implementación el equivalente a 18 años de trabajo individual (realmente no se tardó tanto puesto que el trabajo se desarrolló en equipo). Hasta que la teoría de autómatas y lenguajes formales no se aplicó a la creación de traductores, su desarrollo ha estado plagado de problemas y errores. Sin embargo, hoy día un compilador básico puede ser el proyecto fin de carrera de cualquier estudiante universitario de Informática.

1.2.1 Tipos de traductores

Desde los orígenes de la computación, ha existido un abismo entre la forma en que las personas expresan sus necesidades y la forma en que un ordenador es capaz de interpretar instrucciones. Los traductores han intentado salvar este abismo para facilitarle el trabajo a los humanos, lo que ha llevado a aplicar la teoría de autómatas a diferentes campos y áreas concretas de la informática, dando lugar a los distintos tipos de traductores que veremos a continuación.

1.2.1.1 Traductores del idioma

Traducen de un idioma dado a otro, como por ejemplo del inglés al español. Este tipo de traductores posee multitud de problemas, a saber:

- Necesidad de inteligencia artificial y problema de las frases hechas. El problema de la inteligencia artificial es que tiene mucho de artificial y poco de inteligencia, por lo que en la actualidad resulta casi imposible traducir frases con un sentido profundo. Como anécdota, durante la guerra fría, en un intento por realizar traducciones automáticas del ruso al inglés y viceversa, se puso a prueba un prototipo introduciendo el texto en inglés: "El espíritu es fuerte pero la carne es débil" cuya traducción al ruso se pasó de nuevo al inglés para ver si coincidía con el original. Cual fue la sorpresa de los desarrolladores cuando lo que se obtuvo fue: "El vino está bueno pero la carne está podrida" (en inglés *spirit* significa tanto espíritu como alcohol). Otros ejemplos difíciles de traducir lo constituyen las frases hechas como: "Piel de gallina", "por si las moscas", "molar mazo", etc.
- Difícil formalización en la especificación del significado de las palabras.
- Cambio del sentido de las palabras según el contexto. Ej: "por decir aquello, se llevó una galleta".
- En general, los resultados más satisfactorios en la traducción del lenguaje natural se han producido sobre subconjuntos restringidos del lenguaje. Y aún más, sobre subconjuntos en los que hay muy poco margen de ambigüedad en la interpretación de los textos: discursos jurídicos, documentación técnica, etc.

1.2.1.2 Compiladores

Es aquel traductor que tiene como entrada una sentencia en lenguaje formal y como salida tiene un fichero ejecutable, es decir, realiza una traducción de un código de alto nivel a código máquina (también se entiende por compilador aquel programa que proporciona un fichero objeto en lugar del ejecutable final).

1.2.1.3 Intérpretes

Es como un compilador, solo que la salida es una ejecución. El programa de entrada se reconoce y ejecuta a la vez. No se produce un resultado físico (código máquina) sino lógico (una ejecución). Hay lenguajes que sólo pueden ser interpretados, como p.ej. SNOBOL (*StriNg Oriented SimBOlyc Language*), LISP (*LISt Processing*), algunas versiones de BASIC (*Beginner's All-purpose Symbolic Instruction Code*), etc.

Su principal ventaja es que permiten una fácil depuración. Entre los inconvenientes podemos citar, en primer lugar, la lentitud de ejecución , ya que al ejecutar a la vez que se traduce no puede aplicarse un alto grado de optimización; por ejemplo, si el programa entra en un bucle y la optimización no está muy afinada, las mismas instrucciones se interpretarán y ejecutarán una y otra vez, enlenteciendo la ejecución del programa. Otro inconveniente es que durante la ejecución, el intérprete debe residir en memoria, por lo que consumen más recursos.

Además de que la traducción optimiza el programa acercándolo a la máquina, los lenguajes interpretados tienen la característica de que permiten construir programas que se pueden modificar a sí mismos.

Algunos lenguajes intentan aunar las ventajas de los compiladores y de los intérpretes y evitar sus desventajas; son los lenguajes pseudointerpretados. En estos, el programa fuente pasa por un pseudocompilador que genera un pseudoejecutable. Para ejecutar este pseudoejecutable se le hace pasar por un motor de ejecución que lo interpreta de manera relativamente eficiente. Esto tiene la ventaja de la portabilidad, ya que el pseudoejecutable es independiente de la máquina en que vaya a ejecutarse, y basta con que en dicha máquina se disponga del motor de ejecución apropiado para poder interpretar cualquier pseudoejecutable. El ejemplo actual más conocido lo constituye el lenguaje Java; también son pseudointerpretadas algunas versiones de Pascal y de COBOL (*COmmon Bussiness Oriented Language*). La figura 1.2 muestra los pasos a seguir en estos lenguajes para obtener una ejecución.

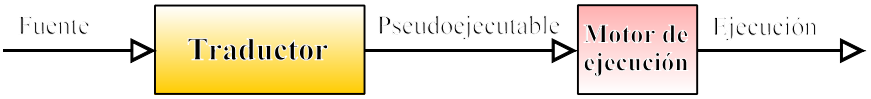

Figure 2 Esquema de traducción/ejecución de un programa interpretado

1.2.1.4 Preprocesadores

Permiten modificar el programa fuente antes de la verdadera compilación. Hacen uso de macroinstrucciones y directivas de compilación. Por ejemplo, en lenguaje C, el preprocesador sustituye la directiva #include Uno.c por el código completo que contiene el fichero "Uno.c", de manera que cuando el compilador comienza su ejecución se encuentra con el código ya insertado en el programa fuente (la figura 1.3 ilustra esta situación). Algunas otras directivas de preprocesamiento permiten compilar trozos de códigos opcionales (lenguajes C y Clipper): #fi, #ifdef, #define, #ifndef, etc. Los preprocesadores suelen actuar de manera transparente para el programador, pudiendo incluso considerarse que son una fase preliminar del compilador.

1.2.1.5 Intérpretes de comandos

Un intérprete de comandos traduce sentencias simples a invocaciones a programas de una biblioteca. Se utilizan especialmente en los sistemas operativos (la *shell* de Unix es un intérprete de comandos). Los programas invocados pueden residir en el *kernel* (núcleo) del sistema o estar almacenados en algún dispositivo externo como rutinas ejecutables que se traen a memoria bajo demanda.

Por ejemplo, si bajo MS-DOS se teclea el comando copy se ejecutará la función de copia de ficheros del sistema operativo, que se encuentra residente en memoria.

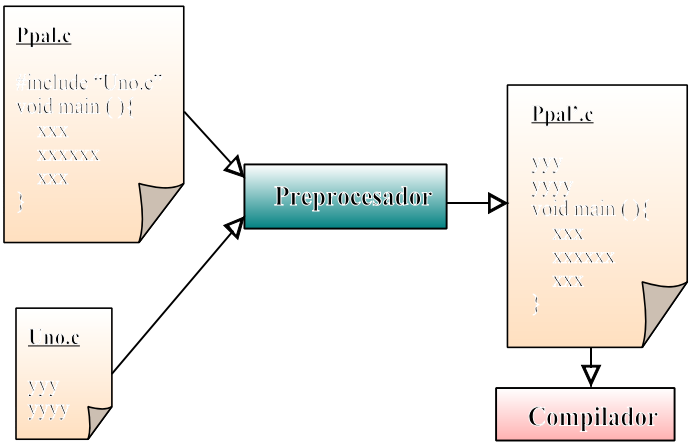

Figure 3 Funcionamiento de la directiva de preprocesamiento #include en lenguaje C

1.2.1.6 Ensambladores y macroensambladores

Son los pioneros de los compiladores, ya que en los albores de la informática, los programas se escribían directamente en código máquina, y el primer paso hacia los lenguajes de alto nivel lo constituyen los ensambladores. En lenguaje ensamblador se establece una relación biunívoca entre cada instrucción y una palabra mnemotécnica, de manera que el usuario escribe los programas haciendo uso de los mnemotécnicos, y el ensamblador se encarga de traducirlo a código máquina puro. De esta manera, los ensambladores suelen producir directamente código ejecutable en lugar de producir ficheros objeto.

Un ensamblador es un compilador sencillo, en el que el lenguaje fuente tiene una estructura tan sencilla que permite la traducción de cada sentencia fuente a una única instrucción en código máquina. Al lenguaje que admite este compilador también se le llama lenguaje ensamblador. En definitiva, existe una correspondencia uno a uno entre las instrucciones ensamblador y las instrucciones máquina. Ej:

Instrucción ensamblador: LD HL, #0100
Código máquina generado: 65h.00h.01h

Por otro lado, existen ensambladores avanzados que permiten definir macroinstrucciones que se pueden traducir a varias instrucciones máquina. A estos programas se les llama macroensambladores, y suponen el siguiente paso hacia los lenguajes de alto nivel. Desde un punto de vista formal, un macroensamblador puede entenderse como un ensamblador con un preprocesador previo.

1.2.1.7 Conversores fuente-fuente

Permiten traducir desde un lenguaje de alto nivel a otro lenguaje de alto nivel,

con lo que se consigue una mayor portabilidad en los programas de alto nivel.

Por ejemplo, si un ordenador sólo dispone de un compilador de Pascal, y queremos ejecutar un programa escrito para otra máquina en COBOL, pues un conversor de COBOL a Pascal solucionará el problema. No obstante el programa fuente resultado puede requerir retoques manuales debido a diversos motivos:

- En situaciones en que el lenguaje destino carece de importantes características que el lenguaje origen sí tiene. Por ejemplo un conversor de Java a C, necesitaría modificaciones ya que C no tiene recolector de basura.
- En situaciones en que la traducción no es inteligente y los programas destino son altamente ineficientes.

1.2.1.8 Compilador cruzado

Es un compilador que genera código para ser ejecutado en otra máquina. Se utilizan en la fase de desarrollo de nuevos ordenadores. De esta manera es posible, p.ej., construir el sistema operativo de un nuevo ordenador recurriendo a un lenguaje de alto nivel, e incluso antes de que dicho nuevo ordenador disponga siquiera de un compilador.

Nótese también que, para facilitar el desarrollo de software de un nuevo ordenador, uno de los primeros programas que se deben desarrollar para éste es, precisamente, un compilador de algún lenguaje de alto nivel.

1.2.2 Conceptos básicos relacionados con la traducción

Vamos a estudiar a continuación diversa terminología relacionada con el proceso de compilación y de construcción de compiladores.

1.2.2.1 Compilación, enlace y carga.

Estas son las tres fases básicas que hay que seguir para que un ordenador ejecute la interpretación de un texto escrito mediante la utilización de un lenguaje de alto nivel. Aunque este libro se centrará exclusivamente en la primera fase, vamos a ver en este punto algunas cuestiones relativas al proceso completo.

Por regla general, el compilador no produce directamente un fichero ejecutable, sino que el código generado se estructura en módulos que se almacenan en un fichero objeto. Los ficheros objeto poseen información relativa tanto al código máquina como a una tabla de símbolos que almacena la estructura de las variables y tipos utilizados por el programa fuente. La figura 1.4 muestra el resultado real que

Figure 4 Entrada y salida de un compilador real

produce un compilador.

Pero, ¿por qué no se genera directamente un fichero ejecutable? Sencillamente, para permitir la compilación separada, de manera que varios programadores puedan desarrollar simultáneamente las partes de un programa más grande y, lo que es más importante, puedan compilarlos independientemente y realizar una depuración en paralelo. Una vez que cada una de estas partes ha generado su correspondiente fichero objeto, estos deben fusionarse para generar un solo ejecutable.

Como se ha comentado, un fichero objeto posee una estructura de módulos también llamados registros. Estos registros tienen longitudes diferentes dependiendo de su tipo y cometido. Ciertos tipos de estos registros almacenan código máquina, otros poseen información sobre las variables globales, y otros incluyen información sobre los objetos externos (p. ej, variables que se supone que están declaradas en otro ficheros. El lenguaje C permite explícitamente esta situación mediante el modificador "extern").

Durante la fase de enlace, el enlazador o *linker* resuelve las referencias cruzadas, (así se llama a la utilización de objetos externos), que pueden estar declarados en otros ficheros objeto, o en librerías (ficheros con extensión "lib" o "dll"), engloba en un único bloque los distintos registros que almacenan código máquina, estructura el bloque de memoria destinado a almacenar las variables en tiempo de ejecución y genera el ejecutable final incorporando algunas rutinas adicionales procedentes de librerías, como por ejemplo las que implementan funciones matemáticas o de e/s básicas. La figura 1.5 ilustra este mecanismo de funcionamiento.

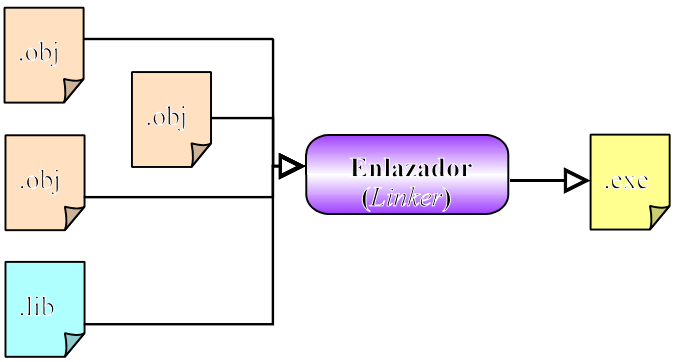

Figure 5 Funcionamiento de un enlazador

De esta manera, el bloque de código máquina contenido en el fichero ejecutable es un código reubicable, es decir, un código que en su momento se podrá ejecutar en diferentes posiciones de memoria, según la situación de la misma en el momento de la ejecución. Según el modelo de estructuración de la memoria del microprocesador, este código se estructura de diferentes formas. Lo más usual es que el fichero ejecutable esté dividido en segmentos: de código, de datos, de pila de datos,

etc.

Cuando el enlazador construye el fichero ejecutable, asume que cada segmento va a ser colocado en la dirección 0 de la memoria. Como el programa va a estar dividido en segmentos, las direcciones a que hacen referencia las instrucciones dentro de cada segmento (instrucciones de cambio de control de flujo, de acceso a datos, etc.), no se tratan como absolutas, sino que son direcciones relativas a partir de la dirección base en que sea colocado cada segmento en el momento de la ejecución. El cargador carga el fichero .exe, coloca sus diferentes segmentos en memoria (donde el sistema operativo le diga que hay memoria libre para ello) y asigna los registros base a sus posiciones correctas, de manera que las direcciones relativas funcionen correctamente. La figura 1.6 ilustra el trabajo que realiza un cargador de programas.

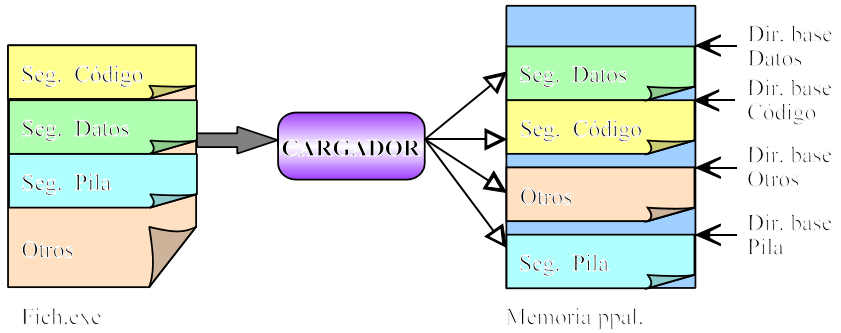

Figure 6 Labor realizada por el cargador. El cargador suele ser parte del sistema operativo

Cada vez que una instrucción máquina hace referencia a una dirección de memoria (partiendo de la dirección 0), el microprocesador se encarga automáticamente de sumar a dicha dirección la dirección absoluta de inicio de su segmento. Por ejemplo para acceder a la variable x almacenada en la dirección 1Fh, que se encuentra en el segmento de datos ubicado en la dirección 8A34h, la instrucción máquina hará referencia a 1Fh, pero el microprocesador la traducirá por 8A34h+1Fh dando lugar a un acceso a la dirección 8A53h: dir absoluta del segmento en memoria + dir relativa de x en el segmento = dir absoluta de x en memoria.

1.2.2.2 Pasadas de compilación

Es el número de veces que un compilador debe leer el programa fuente para generar el código. Hay algunas situaciones en las que, para realizar la compilación, no es suficiente con leer el fichero fuente una sola vez. Por ejemplo, en situaciones en las que existe recursión indirecta (una función A llama a otra B y la B llama a la A). Cuando se lee el cuerpo de A, no se sabe si B está declarada más adelante o se le ha olvidado al programador; o si lo está, si los parámetros reales coinciden en número y tipo con los formales o no. Es más, aunque todo estuviera correcto, aún no se ha

generado el código para B, luego no es posible generar el código máquina correspondiente a la invocación de B puesto que no se sabe su dirección de comienzo en el segmento de código. Por todo esto, en una pasada posterior hay que controlar los errores y rellenar los datos que faltan.

Diferentes compiladores y lenguajes solucionan este problema de maneras distintas. Una solución consiste en hacer dos o más pasadas de compilación, pero ello consume demasiado tiempo puesto que las operaciones de e/s son, hoy por hoy, uno de los puntos fundamentales en la falta de eficiencia de los programas (incluidos los compiladores). Otra solución pasa por hacer una sola pasada de compilación y modificar el lenguaje obligando a hacer las declaraciones de funciones recursivas indirectas antes de su definición. De esta manera, lenguajes como Pascal o Modula-2 utilizan la palabra reservada FORWARD para ello: FORWARD precede la declaración de B, a continuación se define A y por último se define B. Lenguajes como C también permiten el no tener que declarar una función si ésta carece de parámetros y devuelve un entero.

Otros lenguajes dan por implícito el FORWARD, y si aún no se han encontrado aquéllo a que se hace referencia, continúan, esperando que el *linker* resuelva el problema, o emita el mensaje de error.

Actualmente, cuando un lenguaje necesita hacer varias pasadas de compilación, suele colocar en memoria una representación abstracta del fichero fuente, de manera que las pasadas de compilación se realizan sobre dicha representación en lugar de sobre el fichero de entrada, lo que soluciona el problema de la ineficiencia debido a operaciones de e/s.

1.2.2.3 Compilación incremental

Cuando se desarrolla un programa fuente, éste se recompila varias veces hasta obtener una versión definitiva libre de errores. Pues bien, en una compilación incremental sólo se recompilan las modificaciones realizadas desde la última compilación. Lo ideal es que sólo se recompilen aquellas partes que contenían los errores o que, en general, hayan sido modificadas, y que el código generado se reinserte con cuidado en el fichero objeto generado en la última compilación. Sin embargo esto es muy difícil de conseguir y no suele ahorrar tiempo de compilación más que en casos muy concretos.

La compilación incremental se puede llevar a cabo con distintos grados de afinación. Por ejemplo, si se olvida un ‘;’ en una sentencia, se podría generar un fichero objeto transitorio parcial. Si se corrige el error y se añade el ‘;’ que falta y se recompila, un compilador incremental puede funcionar a varios niveles

- A nivel de carácter: se recompila el ‘;’ y se inserta en el fichero objeto la sentencia que faltaba.
- A nivel de sentencia: si el ‘;’ faltaba en la línea 100 sólo se compila la línea 100 y se actualiza el fichero objeto.
- A nivel de bloque: si el ‘;’ faltaba en un procedimiento o bloque sólo se

compila ese bloque y se actualiza el fichero objeto.

- A nivel de fichero fuente: si la aplicación completa consta de 15 ficheros fuente, y solo se modifica 1(al que le faltaba el ';'), sólo se compila aquél al que se le ha añadido el ';', se genera por completo su fichero objeto y luego se enlazan todos juntos para obtener el ejecutable.

Lo ideal es que se hiciese eficientemente a nivel de sentencia, pero lo normal es encontrarlo a nivel de fichero. La mayoría de los compiladores actuales realizan una compilación incremental a este nivel.

Cuando un compilador no admite compilación incremental, suelen suministrar una herramienta externa (como RMAKE en caso de Clipper, MAKE en algunas versiones de C) en la que el programador indica las dependencias entre ficheros, de manera que si se recompila uno, se recompilan todos los que dependen de aquél. Estas herramientas suelen estar diseñadas con un propósito más general y también permiten enlaces condicionales.

1.2.2.4 Autocompilador

Es un compilador escrito en el mismo lenguaje que compila (o parecido). Normalmente, cuando se extiende entre muchas máquinas diferentes el uso de un compilador, y éste se desea mejorar, el nuevo compilador se escribe utilizando el lenguaje del antiguo, de manera que pueda ser compilado por todas esas máquinas diferentes, y dé como resultado un compilador más potente de ese mismo lenguaje.

1.2.2.5 Metacompilador

Este es uno de los conceptos más importantes con los que vamos a trabajar. Un metacompilador es un compilador de compiladores. Se trata de un programa que acepta como entrada la descripción de un lenguaje y produce el compilador de dicho lenguaje. Hoy por hoy no existen metacompiladores completos, pero sí parciales en los que se acepta como entrada una gramática de un lenguaje y se genera un autómata que reconoce cualquier sentencia del lenguaje . A este autómata podemos añadirle código para completar el resto del compilador. Ejemplos de metacompiladores son: Lex, YACC, FLex, Bison, JavaCC, JLex, Cup, PCCTS, MEDISE, etc.

Los metacompiladores se suelen dividir entre los que pueden trabajar con gramáticas de contexto libre y los que trabajan con gramáticas regulares. Los primeros se dedican a reconocer la sintaxis del lenguaje y los segundos trocean los ficheros fuente y lo dividen en palabras.

PCLex es un metacompilador que genera la parte del compilador destinada a reconocer las palabras reservadas. PCYACC es otro metacompilador que genera la parte del compilador que informa sobre si una sentencia del lenguaje es válida o no. JavaCC es un metacompilador que aúna el reconocimiento de palabras reservadas y la aceptación o rechazo de sentencias de un lenguaje. PCLex y PCYACC generan código C y admiten descripciones de un lenguaje mediante gramáticas formales, mientras que JavaCC produce código Java y admite descripciones de lenguajes expresadas en

notación BNF (*Backus-Naur Form*). Las diferencias entre ellas se irán estudiando en temas posteriores.

1.2.2.6 Descompilador

Un descompilador realiza una labor de traducción inversa, esto es, pasa de un código máquina (programa de salida) al equivalente escrito en el lenguaje que lo generó (programa fuente). Cada descompilador trabaja con un lenguaje de alto nivel concreto.

La descompilación suele ser una labor casi imposible, porque al código máquina generado casi siempre se le aplica una optimización en una fase posterior, de manera que un mismo código máquina ha podido ser generado a partir de varios códigos fuente. Por esto, sólo existen descompiladores de aquellos lenguajes en los que existe una relación biyectiva entre el código destino y el código fuente, como sucede con los desensambladores, en los que a cada instrucción máquina le corresponde una y sólo una instrucción ensamblador.

Los descompiladores se utilizan especialmente cuando el código máquina ha sido generado con opciones de depuración, y contiene información adicional de ayuda al descubrimiento de errores (puntos de ruptura, seguimiento de trazas, opciones de visualización de variables, etc.).

También se emplean cuando el compilador no genera código máquina puro, sino pseudocódigo para ser ejecutado a través de un pseudointérprete. En estos casos suele existir una relación biyectiva entre las instrucciones del pseudocódigo y las construcciones sintácticas del lenguaje fuente, lo que permite reconstruir un programa de alto nivel a partir del de un bloque de pseudocódigo.

1.3 Estructura de un traductor

Un traductor divide su labor en dos etapas: una que analiza la entrada y genera estructuras intermedias y otra que sintetiza la salida a partir de dichas estructuras. Por tanto, el esquema de un traductor pasa de ser el de la figura 1.1, a ser el de la figura 1.7.

Básicamente los objetivos de la etapa de análisis son: a) controlar la corrección del programa fuente, y b) generar las estructuras necesarias para comenzar la etapa de síntesis.

Para llevar esto a cabo, la etapa de análisis consta de las siguientes fases:

- ✎ Análisis lexicográfico. Divide el programa fuente en los componentes básicos

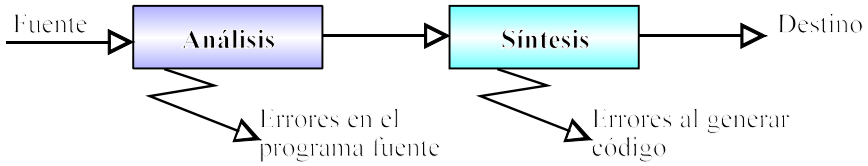

Figure 7 Esquema por etapas de un traductor

del lenguaje a compilar. Cada componente básico es una subsecuencia de caracteres del programa fuente, y pertenece a una categoría gramatical: números, identificadores de usuario (variables, constantes, tipos, nombres de procedimientos, ...), palabras reservadas, signos de puntuación, etc.

- ✎ Análisis sintáctico. Comprueba que la estructura de los componentes básicos sea correcta según las reglas gramaticales del lenguaje que se compila.
- ✎ Análisis semántico. Comprueba que el programa fuente respeta las directrices del lenguaje que se compila (todo lo relacionado con el significado): chequeo de tipos, rangos de valores, existencia de variables, etc.

Cualquiera de estas tres fases puede emitir mensajes de error derivados de fallos cometidos por el programador en la redacción de los textos fuente. Mientras más errores controle un compilador, menos problemas dará un programa en tiempo de ejecución. Por ejemplo, el lenguaje C no controla los límites de un array, lo que provoca que en tiempo de ejecución puedan producirse comportamientos del programa de difícil explicación.

La etapa de síntesis construye el programa objeto deseado (equivalente semánticamente al fuente) a partir de las estructuras generadas por la etapa de análisis. Para ello se compone de tres fases fundamentales:

- ✎ Generación de código intermedio. Genera un código independiente de la máquina muy parecido al ensamblador. No se genera código máquina directamente porque así es más fácil hacer pseudocompiladores y además se facilita la optimización de código independientemente del microprocesador.
- ✎ Generación del código máquina. Crea un bloque de código máquina ejecutable, así como los bloques necesarios destinados a contener los datos.
- ✎ Fase de optimización. La optimización puede realizarse sobre el código intermedio (de forma independiente de las características concretas del microprocesador), sobre el código máquina, o sobre ambos. Y puede ser una aislada de las dos anteriores, o estar integrada con ellas.

1.3.1 Construcción sistemática de compiladores

Con frecuencia, las fases anteriores se agrupan en una **etapa inicial** (*front-end*) y una **etapa final** (*back- end*). La etapa inicial comprende aquellas fases, o partes de fases, que dependen exclusivamente del lenguaje fuente y que son independientes de la máquina para la cual se va a generar el código. En la etapa inicial se integran los análisis léxicos y sintácticos, el análisis semántico y la generación de código intermedio. La etapa inicial también puede hacer cierta optimización de código e incluye además, el manejo de errores correspondiente a cada una de esas fases.

La etapa final incluye aquellas fases del compilador que dependen de la máquina destino y que, en general, no dependen del lenguaje fuente sino sólo del lenguaje intermedio. En esta etapa, se encuentran aspectos de la fase de generación de

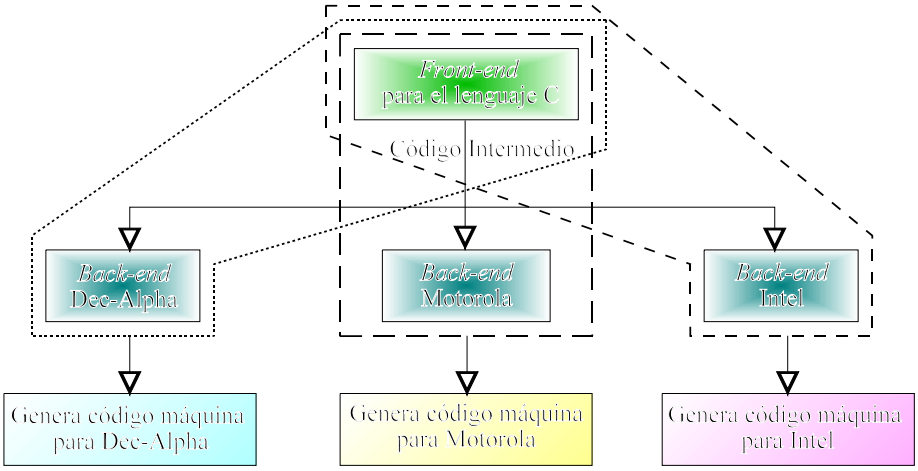

Figure 8 Construcción de tres compiladores de C reutilizando un *front-end*

código, además de su optimización, junto con el manejo de errores necesario y el acceso a las estructuras intermedias que haga falta.

Se ha convertido en una práctica común el tomar la etapa inicial de un compilador y rehacer su etapa final asociada para producir un compilador para el mismo lenguaje fuente en una máquina distinta. También resulta tentador crear compiladores para varios lenguajes distintos y generar el mismo lenguaje intermedio para, por último, usar una etapa final común para todos ellos, y obtener así varios compiladores para una máquina. Para ilustrar esta práctica y su inversa, la figura 1.8

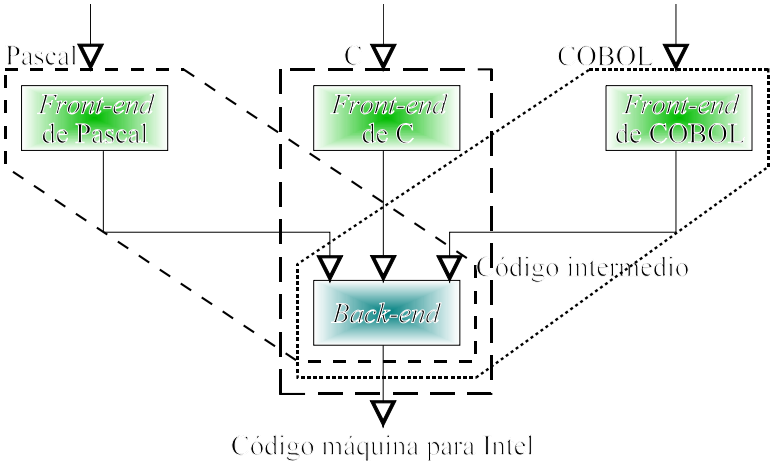

Figure 9 Creación de tres compiladores (Pascal, C y COBOL) para una misma máquina Intel

Muestra una situación en la que se quiere crear un compilador del lenguaje C para tres máquinas diferentes: Dec-Alpha (Unix), Motorola (Mac OS) e Intel (MS-DOS). Cada bloque de líneas punteadas agrupa un *front-end* con un *back-end* dando lugar a un compilador completo.

De manera inversa se podrían construir tres compiladores de Pascal, C y COBOL para una misma máquina, p.ej. Intel, como se ilustra en la figura 1.9.

Por último, la creación de compiladores de Pascal, C y COBOL para las máquinas Dec-Alpha, Motorola e Intel, pasaría por la combinación de los métodos anteriores, tal como ilustra la figura 1.10.

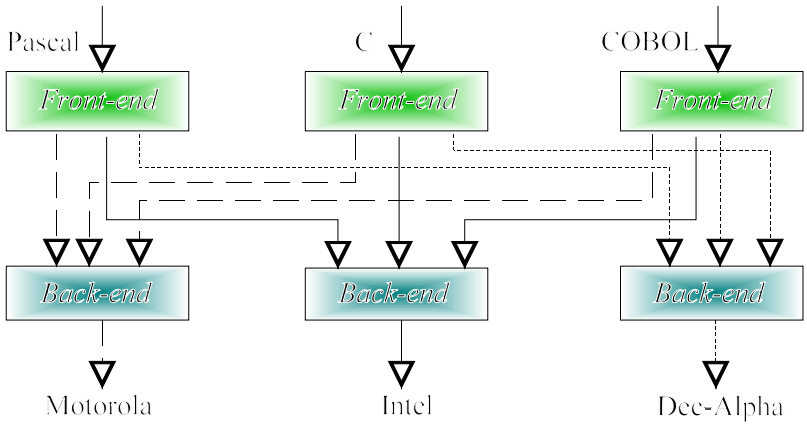

Figure 10 La combinación de cada *front-end* con un *back-end* da lugar a un compilador distinto: tres de Pascal, tres de C y tres de COBOL. El esfuerzo se ha reducido considerablemente.

1.3.2 La tabla de símbolos

Una función esencial de un compilador es registrar los identificadores de usuario (nombres de variables, de funciones, de tipos, etc.) utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador. Estos atributos pueden proporcionar información sobre la memoria asignada a un identificador, la dirección de memoria en que se almacenará en tiempo de ejecución, su tipo, su ámbito (la parte del programa donde es visible), etc.

Pues bien, la tabla de símbolos es una estructura de datos que posee información sobre los identificadores definidos por el usuario, ya sean constantes, variables, tipos u otros. Dado que puede contener información de diversa índole, debe hacerse de forma que su estructura no sea uniforme, esto es, no se guarda la misma información sobre una variable del programa que sobre un tipo definido por el usuario. Hace funciones de diccionario de datos y su estructura puede ser una tabla hash, un árbol binario de búsqueda, etc., con tal de que las operaciones de acceso sean lo

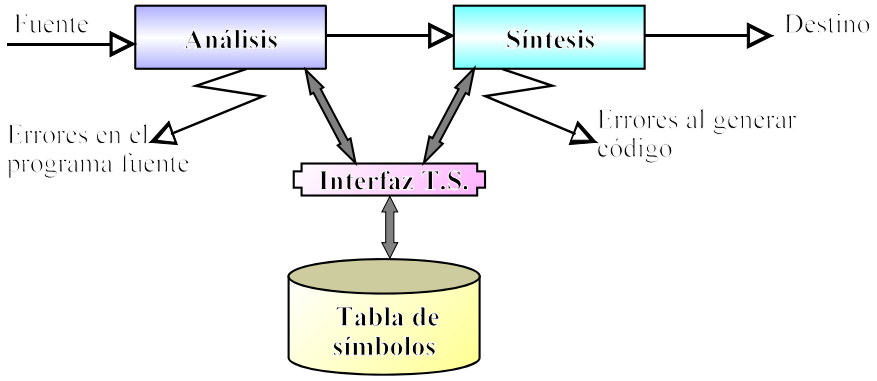

Figure 11 Esquema por etapas definitivo de un traductor

bastante eficientes.

Tanto la etapa de análisis como la de síntesis accede a esta estructura, por lo que se halla muy acoplada al resto de fases del compilador. Por ello conviene dotar a la tabla de símbolos de una interfaz lo suficientemente genérica como para permitir el cambio de las estructuras internas de almacenamiento sin que estas fases deban ser retocadas. Esto es así porque suele ser usual hacer un primer prototipo de un compilador con una tabla de símbolos fácil de construir (y por tanto, ineficiente), y cuando el compilador ya ha sido finalizado, entonces se procede a sustituir la tabla de símbolos por otra más eficiente en función de las necesidades que hayan ido surgiendo a lo largo de la etapa de desarrollo anterior. Siguiendo este criterio, el esquema general definitivo de un traductor se detalla en la figura 1.11. La figura 1.12 ilustra el esquema por fases, donde cada etapa ha sido sustituida por las fases que la componen y se ha hecho mención explícita del preprocesador..

1.4 Ejemplo de compilación

Vamos a estudiar a continuación las diferentes tareas que lleva a cabo cada fase de un compilador hasta llegar a generar el código asociado a una sentencia de C. La sentencia con la que se va a trabajar es la siguiente:

```
#define PORCENTAJE 8
comision = fijo + valor * PORCENTAJE;
```

Para no complicar demasiado el ejemplo, asumiremos que las variables referenciadas han sido previamente declaradas de tipo **int**, e inicializadas a los valores deseados.

Comenzaremos con el preprocesamiento hasta llegar, finalmente, a la fase de generación de código máquina en un microprocesador cualquiera.

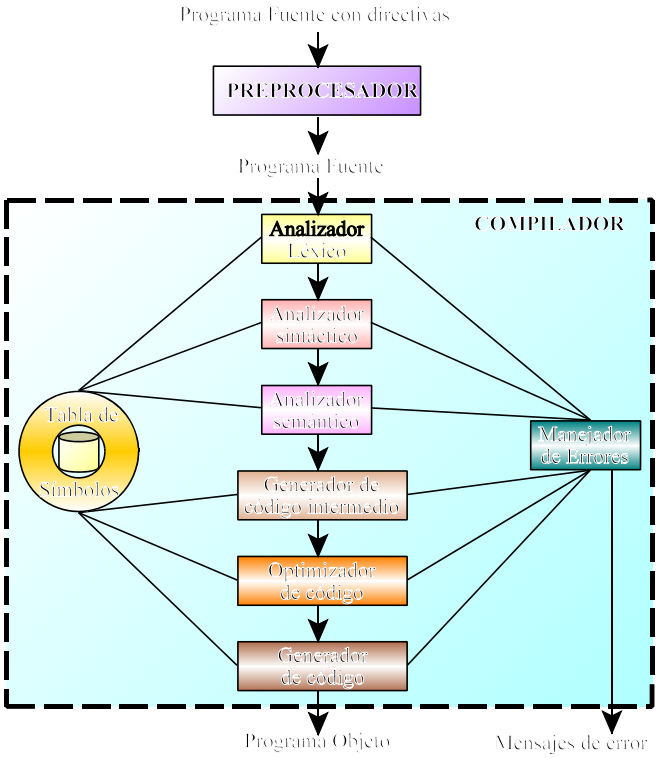

Figure 12 Esquema completo de un compilador por fases con preprocesador.

1.4.1 Preprocesamiento

Como ya hemos visto, el código fuente de una aplicación se puede dividir en módulos almacenados en archivos distintos. La tarea de reunir el programa fuente a menudo se confía a un programa distinto, llamado preprocesador. El preprocesador también puede expandir abreviaturas, llamadas macros, a proposiciones del lenguaje fuente. En nuestro ejemplo, la constante PORCENTAJE se sustituye por su valor, dando lugar al texto:

```
comision = fijo + valor * 8;
```

que pasa a ser la fuente que entrará al compilador.

1.4.2 Etapa de análisis

En esta etapa se controla que el texto fuente sea correcto en todos los sentidos, y se generan las estructuras necesarias para la generación de código.

1.4.2.1 Fase de análisis lexicográfico

En esta fase, la cadena de caracteres que constituye el programa fuente se lee de izquierda a derecha y se agrupa en componentes léxicos, que son secuencias de caracteres que tienen un significado atómico; además el analizador léxico trabaja con la tabla de símbolos introduciendo en ésta los nombres de las variables.

En nuestro ejemplo los caracteres de la proposición de asignación

comision= fijo + valor * 8 ;

se agruparían en los componentes léxicos siguientes:

1.- El identificador comision.
2.- El símbolo de asignación '='.
3.- El identificador fijo.
4.- El signo de suma '+'.
5.- El identificador valor.
6.- El signo de multiplicación '*'.
7.- El número 8.
8.- El símbolo de fin de sentencia ';'.

La figura 1.13 ilustra cómo cada componente léxico se traduce a su categoría gramatical, y se le asocia alguna información, que puede ser un puntero a la tabla de símbolos donde se describe el identificador, o incluso un valor directo, como ocurre en el caso del literal 8. Así, cada componente se convierte en un par (categoría, atributo), y se actualiza la tabla de símbolos. Esta secuencia de pares se le pasa a la siguiente fase de análisis.

Nótese como los espacios en blanco que separan los caracteres de estos componentes léxicos normalmente se eliminan durante el análisis léxico, siempre y cuando la definición del lenguaje a compilar así lo aconseje, como ocurre en C. Lo mismo pasa con los tabuladores innecesarios y con los retornos de carro. Los comentarios, ya estén anidados o no, también son eliminados.

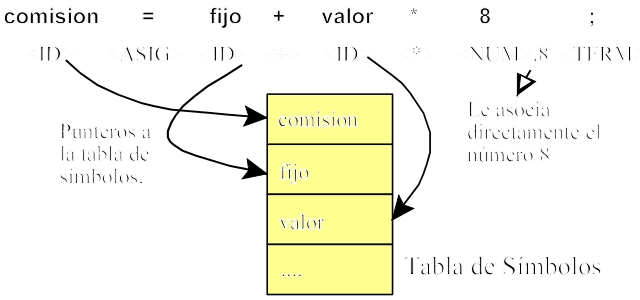

Figure 13 Transformación realizada por el analizador lexicográfico

1.4.2.2 Fase de análisis sintáctico

Trabaja con una gramática de contexto libre y genera el árbol sintáctico que reconoce su sentencia de entrada. En nuestro caso las categorías gramaticales del análisis léxico son los terminales de la gramática. Para el ejemplo que nos ocupa podemos partir de la gramática:

S → <ID> <ASIG> expr <TERM>
expr → <ID>
| <ID> <+> expr
| <ID> <*> expr
| <NUM>

de manera que el análisis sintáctico intenta generar un árbol sintáctico que encaje con la sentencia de entrada. Para nuestro ejemplo, dicho árbol sintáctico existe y es el de la figura 1.14. El árbol puede representarse tal y como aparece en esta figura, o bien

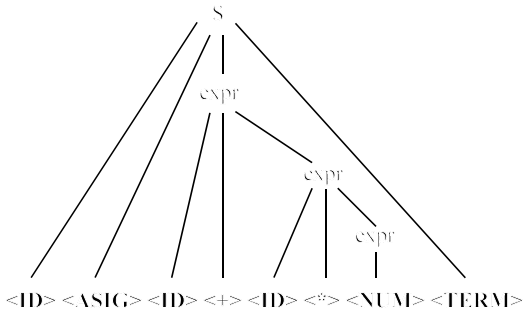

Figure 14Árbol sintáctico de la sentencia de entrada. Aunque sólo se han representado las categorías gramaticales, recuérdese que cada una lleva o puede llevar asociado un atributo.

invertido.

1.4.2.2.1 Compilación dirigida por sintaxis

Se ha representado una situación ideal en la que la fase lexicográfica actúa por separado y, sólo una vez que ha acabado, le suministra la sintáctica su resultado de salida. Aunque proseguiremos en nuestro ejemplo con esta clara distinción entre fases, es importante destacar que el analizador sintáctico tiene el control en todo momento, y el léxico por trozos, a petición del sintáctico. En otras palabras, el sintáctico va construyendo su árbol poco a poco (de izquierda a derecha), y cada vez que necesita un nuevo componente léxico para continuar dicha construcción, se lo solicita al lexicográfico; éste lee nuevos caracteres del fichero de entrada hasta conformar un nuevo componente y, una vez obtenido, se lo suministra al sintáctico, quien continúa la construcción del árbol hasta que vuelve a necesitar otro componente, momento en que se reinicia el proceso. Este mecanismo finaliza cuando se ha obtenido el árbol y ya

no hay más componentes en el fichero de entrada, o bien cuando es imposible construir el árbol.

Esto es tan sólo el principio de lo que se denomina **compilación dirigida por sintaxis** (ver figura 1.15): es aquél mecanismo de compilación en el que el control lo lleva el analizador sintáctico, y todas las demás fases están sometidas a él.

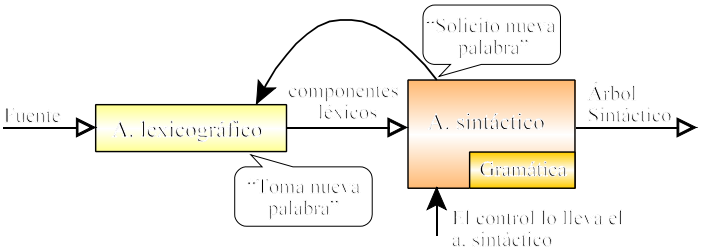

Figure 15 Análisis dirigido por sintaxis

1.4.2.3 Fase de análisis semántico

Esta fase revisa el árbol sintáctico junto con los atributos y la tabla de símbolos para tratar de encontrar errores semánticos. Para todo esto se analizan los operadores y operandos de expresiones y proposiciones. Finalmente reúne la información necesaria sobre los tipos de datos para la fase posterior de generación de código.

El componente más importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si los operandos de cada operador son compatibles según la especificación del lenguaje fuente. Si suponemos que nuestro lenguaje solo trabaja con números reales, la salida de esta fase sería su mismo árbol, excepto porque el atributo de <NUM>, que era el entero 8 a la entrada, ahora pasaría a ser el real 8,0. Además se ha debido controlar que las variables implicadas en la sentencia, a saber, comision, fijo y valor son compatibles con el tipo numérico de la constante 8,0.

1.4.3 Etapa de síntesis

En la etapa anterior se ha controlado que el programa de entrada es correcto. Por tanto, el compilador ya se encuentra en disposición de generar el código máquina equivalente semánticamente al programa fuente. Para ello se parte de las estructuras generadas en dicha etapa anterior: árbol sintáctico y tabla de símbolos.

1.4.3.1 Fase de generación de código intermedio

Después de la etapa de análisis, se suele generar una representación intermedia explícita del programa fuente. Dicha representación intermedia se puede considerar como un programa para una máquina abstracta.

Cualquier representación intermedia debe tener dos propiedades importantes; debe ser fácil de generar y fácil de traducir al código máquina destino. Así, una representación intermedia puede tener diversas formas. En el presente ejemplo se trabajará con una forma intermedia llamada "código de tres direcciones", que es muy parecida a un lenguaje ensamblador para un microprocesador que carece de registros y sólo es capaz de trabajar con direcciones de memoria y literales. En el código de tres direcciones cada instrucción tiene como máximo tres operandos. Siguiendo el ejemplo propuesto, se generaría el siguiente código de tres direcciones:

```
t1 = 8.0
t2 = valor * t1
t3 = fijo + t2
comision = t3
```

De este ejemplo se pueden destacar varias propiedades del código intermedio escogido:

- Cada instrucción de tres direcciones tiene a lo sumo un operador, además de la asignación.
- El compilador debe generar un nombre temporal para guardar los valores intermedios calculados por cada instrucción: t1, t2 y t3.
- Algunas instrucciones tienen menos de tres operandos, como la primera y la última instrucciones del ejemplo.

1.4.3.2 Fase de optimización de código

Esta fase trata de mejorar el código intermedio, de modo que en la siguiente fase resulte un código de máquina más rápido de ejecutar. Algunas optimizaciones son triviales. En nuestro ejemplo hay una forma mejor de realizar el cálculo de la comisión, y pasa por realizar sustituciones triviales en la segunda y cuarta instrucciones, obteniéndose:

```
t2 = valor * 8.0
comision= fijo + t2
```

El compilador puede deducir que todas las apariciones de la variable t1 pueden sustituirse por la constante 8,0, ya que a t1 se le asigna un valor que ya no cambia, de modo que la primera instrucción se puede eliminar. Algo parecido sucede con la variable t3, que se utiliza sólo una vez, para transmitir su valor a comision en una asignación directa, luego resulta seguro sustituir comision por t3, a raíz de lo cual se elimina otra de las líneas del código intermedio.

1.4.3.3 Fase de generación de código máquina

La fase final de un compilador es la generación de código objeto, que por lo general consiste en código máquina reubicable o código ensamblador. Cada una de las variables usadas por el programa se traduce a una dirección de memoria (esto también se ha podido hacer en la fase de generación de código intermedio). Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asignación de variables a

registros.

Siguiendo el mismo ejemplo, y utilizando los registros R1 y R2 de un microprocesador hipotético, la traducción del código optimizado podría ser:

```
MOVE [1Ah], R1
MULT #8.0, R1
MOVE [15h], R2
ADD R1, R2
MOVE R2, [10h]
```

El primer y segundo operandos de cada instrucción especifican una fuente y un destino, respectivamente. Este código traslada el contendido de la dirección [1Ah] al registro R1, después lo multiplica por la constante real 8.0. La tercera instrucción pasa el contenido de la dirección [15h] al registro R2. La cuarta instrucción le suma el valor previamente calculado en el registro R1. Por último el valor del registro R2 se pasa a la dirección [10h]. Como el lector puede suponer, la variable comision se almacena en la dirección [10h], fijo en [15h] y valor en [1Ah].