TEMA 1

INTRODUCCIÓN

1.1. VISIÓN HISTÓRICA DEL DESARROLLO DE LOS COMPILADORES

En 1946 se desarrolló el primer ordenador digital. En un principio, estas máquinas ejecutaban instrucciones consistentes en códigos numéricos que señalan a los circuitos de la máquina los estados correspondientes a cada operación. Esta expresión mediante códigos numéricos se llamó *Lenguaje Máquina*, interpretado por un secuenciador cableado o por un microprograma.

Pero los códigos numéricos de las máquinas son engorrosos. Pronto los primeros usuarios de estos ordenadores descubrieron la ventaja de escribir sus programas mediante claves más fáciles de recordar que esos códigos numéricos; al final, todas esas claves juntas se traducían manualmente a Lenguaje Máquina. Estas claves constituyen los llamados lenguajes ensambladores, que se generalizaron en cuanto se dio el paso decisivo de hacer que las propias máquinas realizaran el proceso mecánico de la traducción. A este trabajo se le llama ensamblar el programa.

Dada su correspondencia estrecha con las operaciones elementales de las máquinas, las instrucciones de los lenguajes ensambladores obligan a programar cualquier función de una manera minuciosa e iterativa. De hecho, normalmente, cuanto menor es el nivel de expresión de un lenguaje de programación, mayor rendimiento se obtiene en el uso de los recursos físicos (hardware).

A pesar de todo, el lenguaje ensamblador seguía siendo el de una máquina, pero más fácil de manejar. Los trabajos de investigación se orientaron entonces hacia la creación de un lenguaje que expresara las distintas acciones a realizar de una manera lo más sencilla posible para el hombre. Así, en 1950, John Backus dirigió una investigación en I.B.M. en un lenguaje algebraico. En 1954 se empezó a desarrollar un lenguaje que permitía escribir fórmulas matemáticas de manera traducible por un ordenador. Le llamaron FORTRAN (FORmulae TRANslator). Fue el primer lenguaje considerado de alto nivel. Se introdujo en 1957 para el uso de la computadora IBM modelo 704. Permitía una programación más cómoda y breve que lo existente hasta ese momento, lo que suponía un considerable ahorro de trabajo. Surgió así por primera vez el concepto de un traductor, como un programa que traducía un lenguaje a otro lenguaje. En el caso particular de que el lenguaje a traducir es un lenguaje de alto nivel y el lenguaje traducido de bajo nivel, se emplea el término *compilador*.

La tarea de realizar un compilador no fue fácil. El primer compilador de FORTRAN tardó 18 años-persona en realizarse y era muy sencillo.

Este desarrollo del FORTRAN estaba muy influenciado por la máquina objeto en la que iba a ser implementado. Como un ejemplo de ello tenemos el hecho de que los espacios en blanco fuesen ignorados, debido a que el periférico que se utilizaba como entrada de programas (una lectora de tarjetas perforadas) no contaba correctamente los espacios en blanco.

Paralelamente al desarrollo de FORTRAN en América, en Europa surgió una corriente más universitaria, que pretendía que la definición de un lenguaje fuese independiente de la máquina y en donde los algoritmos se pudieran expresar de forma más simple.

Esta corriente estuvo muy influida por los trabajos sobre gramáticas de contexto libre publicados por Chomsky dentro de su estudio de lenguajes naturales.

Con estas ideas surgió un grupo europeo encabezado por el profesor F.L.Bauer (de la Universidad de Munich). Este grupo definió un lenguaje de usos múltiples independiente de una realización concreta sobre una máquina. Pidieron colaboración a la asociación americana A.C.M. (Association for Computing Machinery) y se formó un comité en el que participó J. Backus que colaboraba en esta investigación. De esa unión surgió un informe definiendo un International Algebraic Language (I.A.L.), publicado en Zurich en 1958. Posteriormente este lenguaje se llamó ALGOL 58 (ALGOritmic Language). En 1969, el lenguaje fue revisado y llevó a una nueva versión que se llamó ALGOL 60. La versión actual es ALGOL 68, un lenguaje modular estructurado en bloques.

En el ALGOL aparecen por primera vez muchos de los conceptos de los nuevos lenguajes algorítmicos:

- Definición de la sintaxis en notación BNF (Backus-Naur Form).
- Formato libre.
- Declaración explícita de tipo para todos los identificadores.
- Estructuras iterativas más generales.
- Recursividad.
- Paso de parámetros por valor y por nombre.
- Estructura de bloques, lo que determina la visibilidad de los identificadores.

Junto a este desarrollo en los lenguajes, también se iba avanzando en la técnica de compilación. En 1958, Strong y otros proponían una solución al problema de que un compilador fuera utilizable por varias máquinas objeto. Para ello, se dividía por primera vez el compilador en dos fases, designadas como el "*front end*" y el "*back end*".

La primera fase (*front end*) es la encargada de analizar el programa fuente y la segunda fase (*back end*) es la encargada de generar código para la máquina objeto.

El puente de unión entre las dos fases era un lenguaje intermedio que se designó con el nombre de UNCOL (UNiversal Computer Oriented Language).

Para que un compilador fuera utilizable por varias máquinas bastaba únicamente modificar su *back end*. Aunque se hicieron varios intentos para definir el UNCOL, el proyecto se ha quedado simplemente en un ejercicio teórico. De todas formas, la división de un compilador en *front end* y *back end* fue un adelanto importante.

Ya en estos años se van poniendo las bases para la división de tareas en un compilador. Así, en 1959 Rabin y Scott proponen el empleo de autómatas deterministas y no deterministas para el reconocimiento lexicográfico de los lenguajes. Rápidamente se aprecia que la construcción de analizadores léxicos a partir de expresiones regulares es muy útil en la implementación de los compiladores. En 1968, Johnson apunta diversas soluciones. En 1975, con la aparición de LEX surge el concepto de un generador automático de analizadores léxicos a partir de expresiones regulares, basado en el sistema operativo UNIX.

A partir de los trabajos de Chomsky ya citados, se produce una sistematización de la sintaxis de los lenguajes de programación, y con ello un desarrollo de diversos métodos de análisis sintáctico.

Con la aparición de la notación BNF - desarrollada en primer lugar por Backus en 1960 cuando trabajaba en un borrador del ALGOL 60, modificada en 1963 por Naur y formalizada por Knuth en 1964 - se tiene una guía para el desarrollo del análisis sintáctico.

Los diversos métodos de parsers ascendentes y descendentes se desarrollan durante la década de los 60.

En 1959, Sheridan describe un método de parsing de FORTRAN que introducía paréntesis adicionales alrededor de los operandos para ser capaz de analizar las expresiones. Más adelante, Floyd introduce la técnica de la precedencia de operador y el uso de las funciones de precedencia. A mitad de la década de los 60, Knuth define las gramáticas LR y describe la construcción de una tabla canónica de parser LR.

Por otra parte, el uso por primera vez de un parsing descendente recursivo tuvo lugar en el año 1961. En el año 1968 se estudian y definen las gramáticas LL así como los parsers predictivos. También se estudia la eliminación de la recursión a la izquierda de producciones que contienen acciones semánticas sin afectar a los valores de los atributos.

En los primeros años de la década de los 70, se describen los métodos SLR y LALR de parser LR. Debido a su sencillez y a su capacidad de análisis para una gran variedad de lenguajes, la técnica de parsing LR va a ser la elegida para los generadores automáticos de parsers. A mediados de los 70, Johnson crea el generador de analizadores sintácticos YACC para funcionar bajo un entorno UNIX.

Junto al análisis sintáctico, también se fue desarrollando el análisis semántico. En los primeros lenguajes (FORTRAN y ALGOL 60) los tipos posibles de los datos eran muy simples, y la comprobación de tipos era muy sencilla. No se permitía la coerción de tipos, pues ésta era una cuestión difícil y era más fácil no permitirlo.

Con la aparición del ALGOL 68 se permitía que las expresiones de tipo fueran construidas sistemáticamente. Más tarde, de ahí surgió la equivalencia de tipos por nombre y estructural.

El manejo de la memoria como una implementación tipo pila se usó por primera vez en 1958 en el primer proyecto de LISP. La inclusión en el ALGOL 60 de procedimientos recursivos potenció el uso de la pila como una forma cómoda de manejo de la memoria. Dijkstra introdujo posteriormente el uso del *display* para acceso a variables no locales en un lenguaje de bloques.

También se desarrollaron estrategias para mejorar las rutinas de entrada y de salida de un procedimiento. Así mismo, y ya desde los años 60, se estudió el paso de parámetros a un procedimiento por nombre, valor y variable.

Con la aparición de lenguajes que permiten la localización dinámica de datos, se desarrolla otra forma de manejo de la memoria, conocida por el nombre de *heap* (montículo). Se han desarrollado varias técnicas para el manejo del *heap* y los problemas que con él se presentan, como son las referencias perdidas y la recogida de basura.

La técnica de la optimización apareció desde el desarrollo del primer compilador de FORTRAN. Backus comenta cómo durante el desarrollo del FORTRAN se tenía el miedo de que el programa resultante de la compilación fuera más lento que si se hubiera escrito a mano. Para evitar esto, se introdujeron algunas optimizaciones en el cálculo de los índices dentro de un bucle.

Pronto se sistematizan y se recoge la división de optimizaciones independientes de la máquina y dependientes de la máquina. Entre las primeras están la propagación de valores, el arreglo de expresiones, la eliminación de redundancias, etc. Entre las segundas se podría encontrar la

localización de registros, el uso de instrucciones propias de la máquina y el reordenamiento de código.

A partir de 1970 comienza el estudio sistemático de las técnicas del análisis de flujo de datos. Su repercusión ha sido enorme en las técnicas de optimización global de un programa.

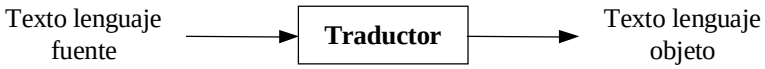
En la actualidad, el proceso de la compilación ya está muy asentado. Un compilador es una herramienta bien conocida, dividida en diversas fases. Algunas de estas fases se pueden generar automáticamente (analizador léxico y sintáctico) y otras requieren una mayor atención por parte del escritor de compiladores (las partes de traducción y generación de código).

De todas formas, y en contra de lo que quizá pueda pensarse, todavía se están llevando a cabo varias vías de investigación en este fascinante campo de la compilación. Por una parte, se están mejorando las diversas herramientas disponibles (por ejemplo, el generador de analizadores léxicos Aardvark para el lenguaje PASCAL). También la aparición de nuevas generaciones de lenguajes -ya se habla de la quinta generación, como de un lenguaje cercano al de los humanos- ha provocado la revisión y optimización de cada una de las fases del compilador.

El último lenguaje de programación de amplia aceptación que se ha diseñado, el lenguaje Java, establece que el compilador no genera código para una máquina determinada sino para una virtual, la *Java Virtual Machine* (JVM), que posteriormente será ejecutado por un intérprete, normalmente incluido en un navegador de Internet. El gran objetivo de esta exigencia es conseguir la máxima portabilidad de los programas escritos y compilados en Java, pues es únicamente la segunda fase del proceso la que depende de la máquina concreta en la que se ejecuta el intérprete.

1.2. ¿QUÉ ES UN COMPILADOR?: TIPOS DE TRADUCTORES

Un *traductor* es cualquier programa que toma como entrada un texto escrito en un lenguaje, llamado fuente y da como salida otro texto en un lenguaje, denominado objeto.



En el caso de que el lenguaje fuente sea un lenguaje de programación de alto nivel y el objeto sea un lenguaje de bajo nivel (ensamblador o código de máquina), a dicho traductor se le denomina *compilador*.

Un *ensamblador* es un compilador cuyo lenguaje fuente es el lenguaje ensamblador.

Un *intérprete* no genera un programa equivalente, sino que toma una sentencia del programa fuente en un lenguaje de alto nivel y la traduce al código equivalente y al mismo tiempo lo ejecuta.

Históricamente, con la escasez de memoria de los primeros ordenadores, se puso de moda el uso de intérpretes frente a los compiladores, pues el programa fuente sin traducir y el intérprete juntos daban una ocupación de memoria menor que la resultante de los compiladores. Por ello los primeros ordenadores personales iban siempre acompañados de un intérprete de BASIC (Spectrum, Commodore VIC-20, PC XT de IBM, etc.). La mejor información sobre los errores por parte del compilador así como una mayor velocidad de ejecución del código resultante hizo que poco a poco se impusieran los compiladores. Hoy en día, y con el problema de la memoria prácticamente resuelto, se puede hablar de un gran predominio de los compiladores frente a los intérpretes, aunque intérpretes como los incluidos en los navegadores de Internet para interpretar el código JVM de Java son la gran excepción.

Ventajas de compilar frente a interpretar:

- Se compila una vez, se ejecuta *n* veces.
- En bucles, la compilación genera código equivalente al bucle, pero interpretándolo se traduce tantas veces una línea como veces se repite el bucle.
- El compilador tiene una visión global del programa, por lo que la información de mensajes de error es mas detallada.

Ventajas del intérprete frente al compilador:

- Un intérprete necesita menos memoria que un compilador. En principio eran más abundantes dado que los ordenadores tenían poca memoria.
- Permiten una mayor interactividad con el código en tiempo de desarrollo.

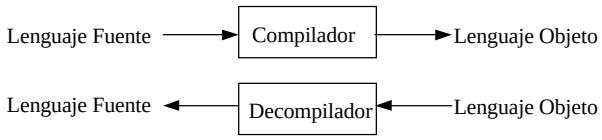
Un compilador no es un programa que funciona de manera aislada, sino que necesita de otros programas para conseguir su objetivo: obtener un programa ejecutable a partir de un programa fuente en un lenguaje de alto nivel. Algunos de esos programas son el preprocesador, el *linker*, el depurador y el ensamblador. El preprocesador se ocupa (dependiendo del lenguaje) de incluir ficheros, expandir macros, eliminar comentarios, y otras tareas similares. El *linker* se encarga de construir el fichero ejecutable añadiendo al fichero objeto generado por el compilador las cabeceras necesarias y las funciones de librería utilizadas por el programa fuente. El depurador permite, si el compilador ha generado adecuadamente el programa objeto, seguir paso a paso la ejecución de un programa. Finalmente, muchos compiladores, en vez de generar código objeto, generan un programa en lenguaje ensamblador que debe después convertirse en un ejecutable mediante un programa ensamblador.

1.2.1. TIPOS DE COMPILADORES

- *Ensamblador*: el lenguaje fuente es lenguaje ensamblador y posee una estructura sencilla.
- *Compilador cruzado*: se genera código en lenguaje objeto para una máquina diferente de la que se está utilizando para compilar. Es perfectamente normal construir un compilador de Pascal que genere código para MS-DOS y que el compilador funcione en Linux y se haya escrito en C++.
- *Compilador con montador*: compilador que compila distintos módulos de forma independiente y después es capaz de enlazarlos.
- *Autocompilador*: compilador que está escrito en el mismo lenguaje que va a compilar. Evidentemente, no se puede ejecutar la primera vez. Sirve para hacer ampliaciones al lenguaje, mejorar el código generado, etc.
- *Metacompilador*: es sinónimo de compilador de compiladores y se refiere a un programa que recibe como entrada las especificaciones del lenguaje para el que se desea obtener un compilador y genera como salida el compilador para ese lenguaje. El desarrollo de los metacompiladores se encuentra con la dificultad de unir la generación de código con la parte de análisis. Lo que sí se han desarrollado son generadores de analizadores léxicos y sintácticos. Por ejemplo, los conocidos:

  LEX $\rightarrow$ generador de analizadores léxicos
  YACC $\rightarrow$ generador de analizadores sintácticos

  desarrollados para UNIX. Los inconvenientes que tienen son que los analizadores que generan no son muy eficientes.
- *Descompilador*: es un programa que acepta como entrada código máquina y lo traduce a un lenguaje de alto nivel, realizando el proceso inverso a la compilación.



Se puede hacer si el programa objeto se guarda también la tabla de símbolos. Si hay optimización de código es imposible descompilar. Hasta ahora no se han obtenido buenos descompiladores, obteniéndose únicamente desensambladores.

1.3. ESTRUCTURA DE UN COMPILADOR

Para la realización del proceso de traducción, es adecuado dividir el compilador en varias fases. Esta división nos ayuda mejor a entender las diversas tareas que debe realizar el compilador y nos permite un estudio independiente y en profundidad de cada una de ellas.

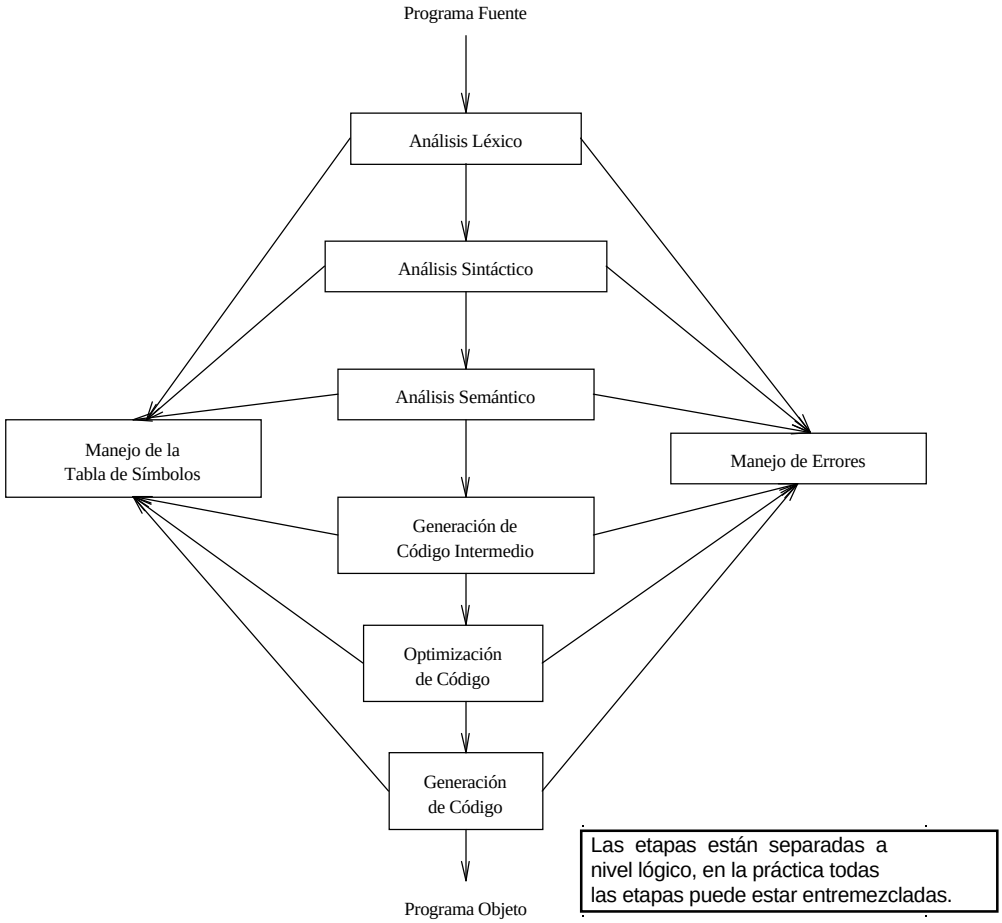


Figura 1.1: Etapas que constituyen el proceso de compilación.

Al principio de la historia de los compiladores, el tamaño del programa ejecutable era un recurso crítico, así como la memoria que utilizaba el compilador para sus datos, por lo que lo frecuente era que cada fase leyera un fichero escrito por la fase anterior y produjera un nuevo fichero con el resultado de las transformaciones realizadas en dicha fase. Esta técnica (inevitable en aquellos tiempos) hacía que el compilador realizara muchas pasadas sobre el programa fuente.

En los últimos años el tamaño del fichero ejecutable de un compilador es relativamente pequeño comparado con el de otros programas del sistema, y además (gracias a los sistemas de

memoria virtual) normalmente no tienen problemas de memoria para compilar un programa medio. Por estos motivos, y dado que escribir y leer un fichero de tamaño similar o mayor que el del programa fuente en cada fase es una pérdida considerable de tiempo (incluso en los sistemas modernos), la tendencia actual es la de reducir el número de ficheros que se leen o escriben y por tanto reducir el número de pasadas, incluso el de aquéllas que se realizan en memoria, sin escribir ni leer nada del disco.

Las fases se agrupan en dos partes o etapas: *front end* (las fases de análisis) y *back end* (las fases de generación y optimización de código). Estas dos etapas se comunican mediante una representación intermedia (generada por el *front end*), que puede ser una representación de la sintaxis del programa (un árbol sintáctico abstracto) o bien puede ser un programa en un lenguaje intermedio. El *front end* depende del lenguaje fuente y casi siempre es independiente (o debe serlo) de la máquina objeto para la que se va a generar código; el *back end* depende del lenguaje objeto y debe ser independiente del lenguaje fuente (excepto quizá para algún tipo de optimización).

Estudiemos ahora cada una de las fases con un poco más de detenimiento.

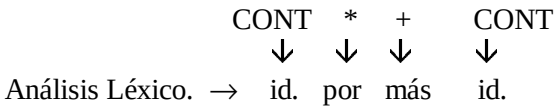
ANÁLISIS LÉXICO

El *analizador léxico*, también conocido como *scanner*, lee los caracteres uno a uno desde la entrada y va formando grupos de caracteres con alguna relación entre sí (*tokens***)**, que constituirán la entrada para la siguiente etapa del compilador. Cada *token* representa una secuencia de caracteres que son tratados como una única entidad. Por ejemplo, en *Pascal* un *token* es la palabra reservada BEGIN, en C: WHILE, etc.

Hay dos tipos de *tokens*: tiras específicas, tales como palabras reservadas (**if**, **while**, **begin**, etc.), el punto y coma, la asignación, los operadores aritméticos o lógicos, etc.; tiras no específicas, como identificadores, constantes o etiquetas.

Se considera que un *token* tiene dos partes componentes: el tipo de *token* y su valor. Las tiras específicas sólo tienen tipo (lo que representan), mientras que las tiras no específicas tienen tipo y valor. Por ejemplo, si "`Contador`" es un identificador, el tipo de *token* será identificador y su valor será la cadena "`Contador`".

El Analizador Léxico es la etapa del compilador que va a permitir saber si es un lenguaje de formato libre o no. Frecuentemente va unido al analizador sintáctico en la misma pasada, funcionando entonces como una subrutina de este último. Ya que es el que va leyendo los caracteres del programa, ignorará aquellos elementos innecesarios para la siguiente fase, como los tabuladores, comentarios, espacios en blanco, etc.



ANÁLISIS SINTÁCTICO

El *analizador sintáctico*, también llamado *parser*, recibe como entrada los *tokens* que le pasa el Analizador Léxico (el analizador sintáctico no maneja directamente caracteres) y comprueba si esos *tokens* van llegando en el orden correcto (orden permitido por el lenguaje). La salida "teórica" de la fase de análisis sintáctico sería un árbol sintáctico.

Así pues, sus funciones son:

- Aceptar lo que es válido sintácticamente y rechazar lo que no lo es.
- Hacer explícito el orden jerárquico que tienen los operadores en el lenguaje de que se trate. Por ejemplo, la cadena A/B*C es interpretada como (A/B)*C en FORTRAN y como A/(B*C) en APL.
- Guiar el proceso de traducción (*traducción dirigida por la sintaxis*).

ANÁLISIS SEMÁNTICO

El análisis semántico es posterior al sintáctico y mucho más difícil de formalizar que éste. Se trata de determinar el tipo de los resultados intermedios, comprobar que los argumentos que tiene un operador pertenecen al conjunto de los operadores posibles, y si son compatibles entre sí, etc. En definitiva, comprobará que el significado de lo que se va leyendo es válido.

La salida "teórica" de la fase de análisis semántico sería un árbol semántico. Consiste en un árbol sintáctico en el que cada una de sus ramas ha adquirido el significado que debe tener.

En el caso de los operadores polimórficos (un único símbolo con varios significados), el análisis semántico determina cuál es el aplicable. Por ejemplo, consideremos la siguiente sentencia de asignación:

A := B + C

En Pascal, el signo “+” sirve para sumar enteros y reales, concatenar cadenas de caracteres y unir conjuntos. El análisis semántico debe comprobar que B y C sean de un tipo común o compatible y que se les pueda aplicar dicho operador. Si B y C son enteros o reales los sumará, si son cadenas las concatenará y si son conjuntos calculará su unión.

EJEMPLO

```
VAR
   ch : CHAR;        (* Un identificador no se puede utilizar si *)
   ent: INTEGER;     (* previamente no se ha definido.          *)
...
ch := ent + 1;       (* En Pascal no es válido, en C sí. *)
```

Análisis Léxico:	Devuelve la secuencia de *tokens*: **id asig id suma numero ptocoma**
Análisis Sintáctico:	Orden de los *tokens* válido
Análisis Semántico:	Tipo de variables asignadas incorrecta

GENERACIÓN DE CÓDIGO INTERMEDIO

Cuando una empresa desarrolla un compilador para un lenguaje fuente y un lenguaje objeto determinados, normalmente no es el único compilador que la empresa piensa desarrollar; es más, muchos fabricantes de microprocesadores tienen una división dedicada a desarrollar compiladores para los nuevos chips que construyen.

Cuando el número de lenguajes fuente crece hasta un número grande *M*, y/o cuando el número de lenguajes objeto también crece hasta un número grande *N*, es necesario encontrar una técnica para evitar tener que diseñar $M \times N$ compiladores. La solución consiste en utilizar un lenguaje intermedio o una representación intermedia; de esta forma sólo hay que construir *M* programas que traduzcan de cada lenguaje fuente al lenguaje intermedio (los *front ends*), y *N* programas que traduzcan del lenguaje intermedio a cada lenguaje objeto (los *back ends*). Desgraciadamente, no existe un único lenguaje intermedio en todos los compiladores, sino que cada empresa que diseña compiladores suele tener su propio lenguaje intermedio. La utilización de un lenguaje intermedio

permite construir en mucho menos tiempo un compilador para otra máquina y también permite construir compiladores para otros lenguajes fuente generando código para la misma máquina.

Por ejemplo, el compilador de C de GNU que se distribuye con Linux es una versión de una familia de compiladores de C para diferentes máquinas o sistemas operativos: Alpha, AIX, Sun, HP, MS-DOS, etc. Además, GNU ha desarrollado un compilador de FORTRAN y otro de Pascal que, al utilizar el mismo lenguaje intermedio, pueden ser portados a todos los sistemas y máquinas en las que ya exista un compilador de C de GNU con relativamente poco esfuerzo.

La generación de código intermedio transforma un árbol de análisis sintáctico (semántico) en una representación en un lenguaje intermedio, que suele ser código suficientemente sencillo para poder luego generar código máquina.

Una forma de hacer esto es mediante el llamado *código de tres direcciones*. Una sentencia en código de tres direcciones es *A* := *B op C*, donde *A*, *B* y *C* son operandos y *op* es un operador binario. También permiten condicionales simples y saltos. Por ejemplo, para la siguiente sentencia:

```
WHILE (A>B) AND (A<=2*B-5) DO A:=A+B
```

el código intermedio generado (código en tres direcciones) será:

```
L1: IF A>B GOTO L2
    GOTO L3
L2: T1 := 2*B         (*nivel más alto que ensamblador*)
    T2 := T1-5        (*pero más sencillo que Pascal*)
    IF A<=T2 GOTO L4
    GOTO L3
L4: A := A+B
    GOTO L1
L3: .....
```

OPTIMIZACIÓN DE CÓDIGO

Esta fase se añade al compilador para conseguir que el programa objeto sea más rápido en la ejecución y que necesite menos memoria a la hora de ejecutarse. El término optimización no es correcto, ya que nunca podemos asegurar que conseguimos un programa óptimo. Con una buena optimización, el tiempo de ejecución puede llegar a reducirse a la mitad, aunque hay máquinas que pueden no llevar esta etapa y generar directamente código máquina. Ejemplos de posibles optimizaciones locales:

- cuando hay dos saltos seguidos, se puede quedar uno solo. El fragmento de programa del ejemplo anterior quedaría así:

```
L1: IF A<=B GOTO L3   (*)
    T1:= 2*B
    T2:= T1-5
    IF A>T2 GOTO L3   (*)
    A:= A+B
    GOTO L1
L3: .....
```

(*): Ahora 2 instrucciones a nivel máquina, antes eran 3: supone un ahorro en tiempo y espacio.

- Eliminar expresiones comunes en favor de una sola expresión. Por ejemplo:

```
A:=B+C+D
E:=B+C+F
```

se convierte en:

```
T1:= B+C
A:= T1+D
E:= T1+F
```

- Optimización de bucles y lazos. Se trata de sacar de los bucles y lazos las expresiones que sean invariantes dentro de ellos. Por ejemplo, dado el siguiente código:

```
REPEAT
   B:= 1
   A:= A-B
UNTIL A=0
```

se trata de sacar del bucle la asignación `B:=1`.

GENERACIÓN DE CÓDIGO

En esta parte el código intermedio optimizado es traducido a una secuencia de instrucciones en ensamblador o en el código de máquina del procesador que nos interese. Por ejemplo, la sentencia `A:=B+C` se convertirá en:

```
LOAD  B
ADD  C
STORE  A
```

suponiendo que estas instrucciones existan de esta forma en el ordenador de que se trate.

Una conversión tan directa produce generalmente un programa objeto que contiene muchas cargas (*loads*) y almacenamientos (*stores*) redundantes, y que utiliza los recursos de la máquina de forma ineficiente. Existen técnicas para mejorar esto, pero son complejas. Una, por ejemplo, es tratar de utilizar al máximo los registros de acceso rápido que tenga la máquina. Así, en el procesador 8086 tenemos los registros internos AX, BX, CX, DX, etc. y podemos utilizarlos en vez de direcciones de memoria.

TABLA DE SÍMBOLOS

Un compilador necesita guardar y usar la información de los objetos que se va encontrando en el texto fuente, como variables, etiquetas, declaraciones de tipos, etc. Esta información se almacena en una estructura de datos interna conocida como tabla de símbolos.

El compilador debe desarrollar una serie de funciones relativas a la manipulación de esta tabla como insertar un nuevo elemento en ella, consultar la información relacionada con un símbolo, borrar un elemento, etc. Como se tiene que acceder mucho a la tabla de símbolos los accesos deben ser lo más rápidos posible para que la compilación sea eficiente.

MANEJO DE ERRORES

Es una de las misiones más importantes de un compilador, aunque, al mismo tiempo, es lo que más dificulta su realización. Donde más se utiliza es en las etapas de análisis sintáctico y semántico, aunque los errores se pueden descubrir en cualquier fase de un compilador. Es una tarea difícil, por dos motivos:

- A veces unos errores ocultan otros.
- A veces un error provoca una avalancha de muchos errores que se solucionan con el primero.

Es conveniente un buen manejo de errores, y que el compilador detecte todos los errores que tiene el programa y no se pare en el primero que encuentre. Hay, pues, dos criterios a seguir a la hora de manejar errores:

- Pararse al detectar el primer error.
- Detectar todos los errores de una pasada.

En el caso de un compilador interactivo (dentro de un entorno de desarrollo integrado, como Turbo-Pascal o Borland C++) no importa que se pare en el primer error detectado, debido a la rapidez y facilidad para la corrección de errores.

1.4. ¿CÓMO SE ESPECIFICA UN COMPILADOR?

Cuando se va a diseñar un compilador, puesto que se trata de un traductor entre dos lenguajes, es necesario especificar el lenguaje fuente y el lenguaje objeto, así como el sistema operativo sobre el que va a funcionar el compilador y el lenguaje utilizado para construir el compilador. Existen algunas herramientas formales para especificar los lenguajes objeto, pero no existe ninguna herramienta cuyo uso sea extendido.

La especificación del lenguaje fuente se divide en tres partes:

- Especificación léxica: en esta parte se especifican los componentes léxicos (*tokens*) o palabras del lenguaje; para ello se utilizan expresiones regulares. A partir de estas expresiones regulares se construye el analizador léxico del compilador.
- Especificación sintáctica: en esta parte se detalla la forma o estructura (la sintaxis) que tendrán los programas en este lenguaje fuente. Para esta tarea se utiliza una gramática independiente del contexto en notación BNF o un diagrama sintáctico que posteriormente se convierte en una gramática; a partir de la gramática se construye el analizador sintáctico del compilador.
- Especificación semántica: en esta parte se describe el significado de cada construcción sintáctica y las reglas semánticas que deben cumplirse; aunque existen notaciones formales para especificar la semántica de un lenguaje, normalmente se especifica con palabras (lenguaje natural). Algunas veces es posible recoger en la especificación sintáctica algunas partes (restricciones, asociatividad y precedencia, etc.) de la especificación semántica. El analizador semántico y el generador de código intermedio se construyen a partir de la especificación semántica.

1.5 APLICACIONES DE ESTAS TÉCNICAS

La importancia práctica de los traductores de lenguajes en la informática se manifiesta principalmente en el uso cotidiano que hace el profesional informático de compiladores e intérpretes, consustancial a la gestión y programación de los sistemas informáticos. Así pues, un conocimiento acerca del funcionamiento interno de estas herramientas básicas resulta fundamental. Pero los conocimientos adquiridos en su estudio encuentran aplicación fuera del campo de la compilación.

Ya Nicklaus Wirth apuntó la importancia del estudio del desarrollo de compiladores para el ingeniero en informática, puesto que son programas muy complejos y que, por tanto, requieren de una técnica de programación disciplinada y estructurada, y porque además desarrolla la visión recursiva del programador, inherente a la compilación. Sólo por esos motivos ya es interesante el estudio de estas técnicas. Por otro lado, es probable que pocas personas con esta formación tengan que realizar o mantener un compilador para un lenguaje de programación, pero muchos

pueden obtener provecho del uso de un gran número de sus técnicas para el diseño de *software* en general.

En efecto, entre los campos de la informática en los que encuentran aplicación las técnicas aprendidas en Compiladores e Intérpretes se pueden citar los siguientes:

- Desarrollo de interfaces textuales. Cualquier programa cuya interacción con el usuario sea algo más que pulsar teclas de opciones o pinchar aquí o allá con el ratón necesitará de estas técnicas para interpretar comandos o cualquier tipo de diálogo hombre-máquina.
- Tratamiento de ficheros de texto con información estructurada. Lenguajes como Perl y Tcl, o comandos como el sed o egrep de UNIX, incorporan tratamiento de expresiones regulares para la detección y/o modificación de patrones en textos.
- Procesadores de texto. Procesadores como vi o Emacs incorporan también la posibilidad de efectuar búsquedas y sustituciones mediante expresiones regulares. Existen también procesadores (entre ellos el Emacs) capaces de analizar y tratar ficheros de textos de organización compleja.
- Diseño e interpretación de lenguajes para el formateo de texto y descripción de gráficos. Sistemas de formateo de texto (como el HTML o el TeX) o para la especificación de tablas (tbl), ecuaciones (eqn), gráficos (Postscript), etc. requieren sofisticados macroprocesadores.
- Gestión de bases de datos. Las técnicas que estamos considerando pueden explotarse tanto en la exploración y proceso de ficheros de información como en la realización de la interface de usuario.
- Procesamiento del lenguaje natural. Las primeras fases de cualquier manipulación de textos escritos en lenguaje natural son las de análisis léxico y sintáctico. Más aún nos acercamos a las técnicas propias de la compilación cuando hablamos de traducción automática.
- Traducción de formatos de ficheros. Si conocemos la estructura de los datos de ficheros con registros de programas obsoletos podremos, utilizando técnicas de traducción propias de la compilación, actualizarlos a formatos actualizados.
- Cálculo simbólico. Usar fórmulas en vez de números.
- Reconocimiento de formas. Las técnicas de análisis sintáctico son ampliamente utilizadas en la detección de patrones en textos, el reconocimiento automático del habla o la visión por computador.