Del Funcionamiento y comportamiento de los Compiladores

Miguel Ángel
superhacker77@hotmail.com

1. Introducción

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 lalocalización 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.

¿Qué es un compilador?
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.

Compilador
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.

2. Clasificación de Compiladores

El programa compilador traduce las instrucciones en un lenguaje de alto nivel a instrucciones que la computadora puede interpretar y ejecutar. Para cada lenguaje de programación se requiere un compilador separado. El compilador traduce todo el programa antes de ejecutarlo. Los compiladores son, pues, programas de traducción insertados en la memoria por el sistema operativo para convertir programas de cómputo en pulsaciones electrónicas ejecutables (lenguaje de máquina). Los compiladores pueden ser de:

- una sola pasada: examina el código fuente una vez, generando el código o programa objeto.
- pasadas múltiples: requieren pasos intermedios para producir un código en otro lenguaje, y una pasada final para producir y optimizar el código producido durante los pasos anteriores.
- Optimación: lee un código fuente, lo analiza y descubre errores potenciales sin ejecutar el programa.
- Compiladores incrementales: generan un código objeto instrucción por instrucción (en vez de hacerlo para todo el programa) cuando el usuario teclea cada orden individual. El otro tipo de compiladores requiere que todos los enunciados o instrucciones se compilen conjuntamente.
- 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:
  generador de analizadores léxicos LEX:
  generador de YACC: 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.

3. Funciones de un compilador

A grandes rasgos un compilador es un programa que lee un programa escrito es un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto. Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente.
A primera vista, la diversidad de compiladores puede parecer abrumadora. Hay miles de lenguajes fuente, desde los lenguajes de programación tradicionales, como FORTRAN o Pascal, hasta los lenguajes especializados que han surgido virtualmente en todas las áreas de aplicación de la informática. Los lenguajes objeto son igualmente variados; un lenguaje objeto puede ser otro lenguaje de programación o el lenguaje de máquina de cualquier computador entre un microprocesador y un supercomputador. A pesar de existir una aparente complejidad por la clasificación de los compiladores, como se vio en el tema anterior, las tareas básicas que debe realizar cualquier compilador son esencialmente las mismas. Al comprender tales tareas, se pueden construir compiladores para una gran diversidad de lenguajes fuente y máquinas objeto utilizando las mismas técnicas básicas.
Nuestro conocimiento sobre cómo organizar y escribir compiladores ha aumentado mucho desde que comenzaron a aparecer
los primeros compiladores a principios de los años cincuenta. Es difícil dar una fecha exacta de la aparición del primer compilador, porque en un principio gran parte del trabajo de experimentación y aplicación se realizó de manera independiente por varios grupos. Gran parte

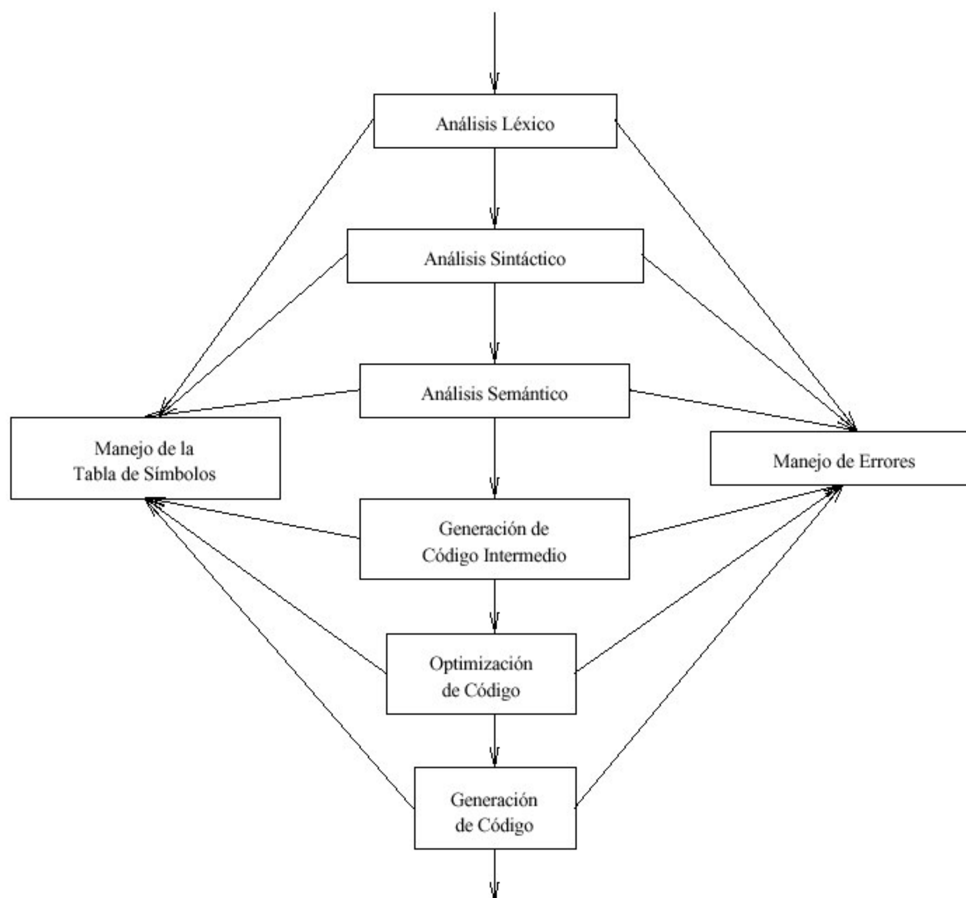
de los primeros trabajos de compilación estaba relacionada con la traducción de fórmulas aritméticas a código de máquina.

En la década de 1950, se consideró a los compiladores como programas notablemente difíciles de escribir. EL primer compilador de FORTRAN, por ejemplo, necesitó para su implantación de 18 años de trabajo en grupo (Backus y otros [1975]). Desde entonces, se han descubierto técnicas sistemáticas para manejar muchas de las importantes tareas que surgen en la compilación. También se han desarrollado buenos lenguajes de implantación, entornos de programación y herramientas de software. Con estos avances, puede hacerse un compilador real incluso como proyecto de estudio en un curso de un semestre sobre diseño sobre de compiladores.

4. Partes en las que trabaja un compilador

Conceptualmente un compilador opera en fases. Cada una de las cuales transforma el programa fuente de una representación en otra. En la figura 3 se muestra una descomposición típica de un compilador. En la práctica se pueden agripar fases y las representaciones intermedias entres las fases agrupadas no necesitan ser construidas explícitamente.

Programa fuente



Programa objeto
Figura 3.- Fases de un compilador.
Las tres primeras fases, que forman la mayor parte de la porción de análisis de un compilador se analizan en la sección IX. Otras dos actividades, la administración de la tabla se símbolos y el manejo de errores, se muestran en interacción con las seis fases de análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, optimación de código y generación de código. De modo informal, también se llamarán "fases" al administrador de la tabla de símbolos y al manejador de errores.

Administrador de la tabla de símbolos
Una función esencial de un compilador es registrar los identificadores 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, su tipo, su ámbito (la parte del programa donde tiene validez) y, en el caso de nombres de procedimientos, cosas como el número y tipos de sus argumentos, el método de pasar cada argumento (por ejemplo, por referencia) y el tipo que devuelve, si los hay.
Una tabla de símbolos es una estructura de datos que contiene un registro por cada identificador, con los campos para los atributos del identificador. La estructura de datos permite encontrar rápidamente el registro de cada identificador y almacenar o consultar rápidamente datos en un registro
Cuando el analizador léxico detecta un indentificador en el programa fuente, el identificador se introduce en la tabla de símbolos. Sin embargo, normalmente los atributos de un identificador no se pueden determinar durante el análisis léxico. Por ejemplo, en una declaración en Pascal como
var posición, inicial, velocidad : real;
El tipo real no se conoce cuando el analizador léxico encuentra posición, inicial y velocidad.
Las fases restantes introducen información sobre los identificadores en la tabla de símbolos y después la utilizan de varias formas. Por ejemplo, cuando se está haciendo el análisis semántico y la generación de código intermedio, se necesita saber cuáles son los tipos de los identificadores, para poder comprobar si el programa fuente los usa de una forma válida y así poder generar las operaciones apropiadas con ellos. El generador de código, por lo general, introduce y utiliza información detallada sobre la memoria asignada a los identificadores.

Detección e información de errores
Cada frase puede encontrar errores. Sin embargo, después de detectar un error. Cada fase debe tratar de alguna forma ese error, para poder continuar la compilación, permitiendo la detección de más errores en el programa fuente. Un compilador que se detiene cuando encuentra el primer error, no resulta tan útil como debiera.
Las fases de análisis sintáctico y semántico por lo general manejan una gran proporción de los errores detectables por el compilador. La fase léxica puede detectar errores donde los caracteres restantes de la entrada no forman ningún componente léxico del lenguaje. Los errores donde la cadena de componentes léxicos violan las reglas de estructura (sintaxis) del lenguaje son determinados por la fase del análisis sintáctico.
Durante el análisis semántico el compilador intenta detectar construcciones que tengan la estructura sintáctica correcta, pero que no tengan significado para la operación implicada, por ejemplo, si se intenta sumar dos identificadores. Uno de los cuales es el nombre de una matriz, y el otro, el nombre de un procedimiento.

Las fases de análisis
Conforme avanza la traducción, la representación interna del programa fuente que tiene el compilador se modifica. Para ilustrar esas representaciones, considérese la traducción de la proposición
Posición := inicial + velocidad * 60 (1)
La figura 4 muestra la representación de esa proposición después de cada frase.
Posición := inicial + velocidad * 60
$Id_1 := id_2 + id_3 * 60$

Tabla de símbolos
1
2
3
4 : =
id_1 +
id_2 *
id_3 entareal
60
temp1 := entareal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
temp1 := id3 * 60.0
id1 := id2 + temp1
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

Figura 4.- Representación de una proposición.
La fase de análisis léxico lee los caracteres de un programa fuente y los agrupa en una cadena de componentes léxicos en los que cada componente representa una secuencia lógicamente coherente de caracteres, como un identificador, una palabra clave (if, while, etc), un carácter de puntuación, o un operador de varios caracteres, como :=. La secuencia de caracteres que forman un componente léxico se denomina lexema del componente.
A ciertos componentes léxicos se les agregará un "valor léxico". Así, cuando se encuentra un identificador como velocidad, el analizador léxico no sólo genera un componente léxico, por ejemplo, id, sino que también introduce el lexema velocidad en la tabla de símbolos, si aún no estaba allí. El valor léxico asociado con esta aparición de id señala la entrada de la tabla de símbolos correspondiente a velocidad.
Usaremos id_1 , id_2 e id_3 para posición, inicial y velocidad, respectivamente, para enfatizar que la representación interna de un identificador es diferente de la secuencia de caracteres que forman el identificador. Por tanto, la representación de (1) después del análisis léxico queda sugerida por:
$id_1 := id_2 + id_3 * 60$ (2)

Se deberían construir componentes para el operador de varios caracteres := y el número 60, para reflejar su representación interna. En la sección IX ya se introdujeron las fases segunda y tercera: los análisis sintáctico y semántico. El análisis sintáctico impone una estructura jerárquica a la cadena de componentes léxicos, que se representará por medio de árboles sintácticos, como se muestra en la figura 5A). Una estructura de datos típica para el árbol se muestra en la figura 5B), en la que un nodo interior es un registro con un campo para el operador y dos campos que contienen apuntadores a los registros de los hijos izquierdo y derecho. Una hoja es un registro con dos o más campos, uno para identificar el componente léxico de la hoja, y los otros para registrar información sobre el componente léxico. Se puede tener información adicional sobre las construcciones del lenguaje añadiendo más campos a les registros de los nodos.
: =
id_1 +
id_2 *
id_3 60
(a)

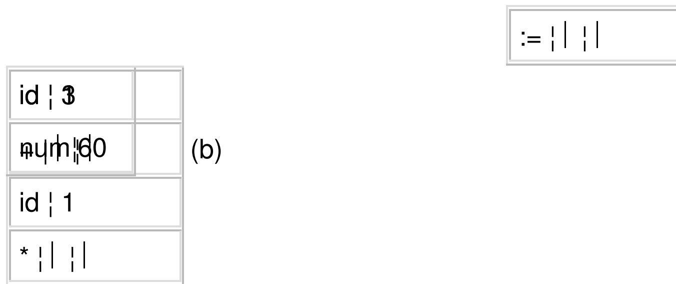


Figura 5.- La estructura de datos en (b) corresponde al árbol en (a).

Generación de código intermedio
Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes; debe ser fácil de producir y fácil de traducir al programa objeto.
La representación intermedia puede tener diversas formas. Existe una forma intermedia llamada "código de tres direcciones", que es como el lenguaje ensamblador para una máquina en la que cada posición de memoria puede actuar como un registro. El código de tres direcciones consiste en una secuencia de instrucciones, cada una de las cuales tiene como máximo tres operandos. El programa fuente de (1) puede aparecer en código de tres direcciones como
temp1 := entarea1(60)
temp2 := id3 * temp1 (2)
temp3 := id2 + temp2
id1 := temp3

Esta representación intermedia tiene varias propiedades. Primera, cada instrucción de tres direcciones tiene a lo sumo un operador, además de la asignación. Por tanto, cuando se generan esas instrucciones el compilador tiene que decidir el orden en que deben efectuarse, las operaciones; la multiplicación precede a la adición al programa fuente de (1).
Segunda, el compilador debe generar un nombre temporal para guardar los valores calculados por cada instrucción. Tercera, algunas instrucciones de "tres direcciones" tienen menos de tres operadores, por ejemplo la primera y la última instrucciones de (2).

Optimación de Código
La fase de optimación de código trata de mejorar el código intermedio de modo que resulte un código de máquina más rápido de ejecutar. Algunas optimaciones son triviales. Por ejemplo, un algoritmo natural genera el código intermedio (2) utilizando una instrucción para cada operador de la representación del árbol después del análisis semántico, aunque hay una forma mejor de realizar los mismos cálculos usando las dos instrucciones
Temp1 := id3 * 60.0 (3)
Id1 := id2 + temp1
Este sencillo algoritmo no tiene nada de malo, puesto que el problema se puede solucionar en la fase de optimación de código. Esto es, el compilador puede deducir que la conversión de 60 de entero a real se puede hacer de una vez por todas en el momento de la compilación, de modo que la operación entreal se puede eliminar. Además, temp3 se usa sólo una vez, para transmitir su valor a id1. Entonces resulta seguro sustituir a id1 por temp3, a partir de lo cual la última proposición de (2) no se necesita y se obtiene el código de (3).
Hay muchas variaciones en la cantidad de optimación de código que ejecutan los distintos compiladores. En lo que hacen mucha optimación llamados "compiladores optimadores", una parte significativa del tiempo del compilador se ocupa en esta fase. Sin embargo hay

optimaciones sencillas que mejoran significativamente del tiempo del compilador se ocupa en esta fase. Sin embargo, hay optimaciones sencillas que mejoran sensiblemente el tiempo de ejecución del programa objeto sin retardar demasiado la compilación.

5. Forma de examinar de un compilador

En la compilación hay dos partes: Análisis y Síntesis. La parte del análisis divide al programa fuente en sus elementos componentes y crea una representación intermedia. De las dos partes, la síntesis es la que requiere la técnica más especializada.
Durante el análisis se determina las operaciones que implica el programa fuente y se registra en una estructura jerárquica llamada árbol. A menudo, se usa una clase especial de árbol llamado árbol sintáctico, donde cada nodo representa una operación y los hijos de un nodo son los argumentos de la operación. Por ejemplo, en la figura 5 se muestra un árbol sintáctico para una proposición de asignación.

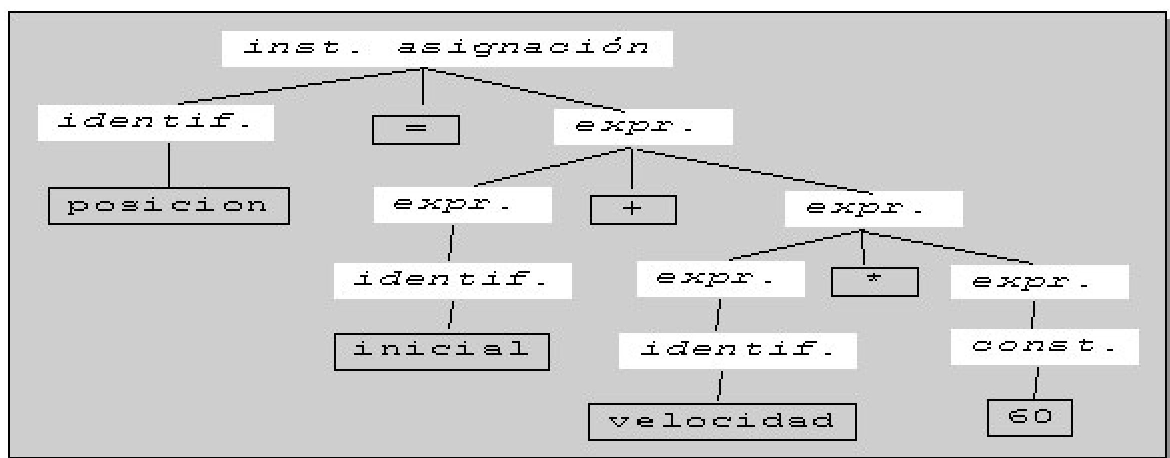


Figura 5. Árbol sintáctico para posición := inicial + velocidad * 60

6. Como se sintetiza el código objeto un compilador estándar, teórica y gráficamente 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.

7. Árboles sintácticos para representar como sintetiza el código objeto un compilador

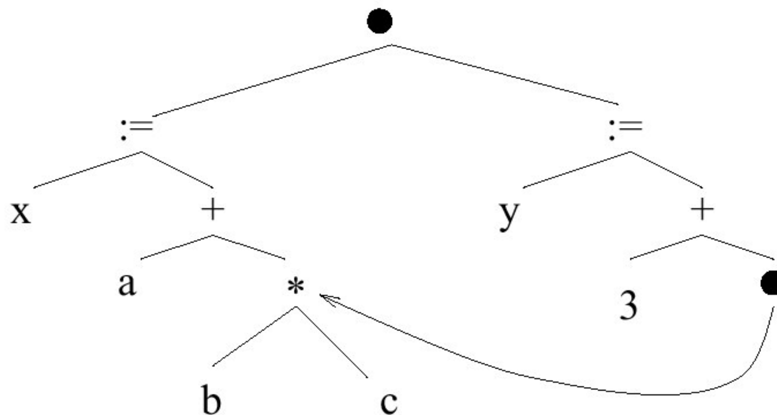


Figura 6.- Generación de código.

```
Push   a       ; a → pila
Push   b       ; b → pila
Load   (c),R1  ; c → R1
Mult   (S),R1  ; b*c → R1
Store  R1,R2   ; R1 → R2
Add    (S),R1  ; a+b*c → R1
Store  R1,(x)  ; R1 → x
Add    #3,R2   ; 3+b*c → R2
Store  R2,(y)  ; R2 → y
```

figura 7.- Código en ensamblador para la figura 6.

8. Herramientas que muestran tipos de análisis de programas fuente

Muchas herramientas de software que manipulan porgraas fuente realizan primero algún tipo de análisis. Algunos ejemplos de tales herramientas son:

1. Editores de estructuras: un editor de estructuras toma como entrada uan secuencia de ordenes para construir un programa fuente. El editor de estructuras no sólo realiza las funciones de creación y modificación de textos de un editor de textos ordinario, sino que también analiza el texto del programa imponiendo al programa fuente una estructura jerárquica apropiada. De esa manera, el editor de estructuras puede realizar tareas adicionales útiles para la preparación de programas. Por ejemplo, puede comprobar si la entrada está formada correctamente, puede proporcionar palabras clave de manera automática y puede saltar desde un begin o un paréntesis izquierdo hasta su correspondiente end o paréntesis derecho. Además, la salida de tal editor suele ser similar a la salida de la fase del análisis de un compilador.
2. Impresoras estéticas: Una impresora estética analiza un programa y lo imprime de forma que la estructura del programa resulte claramente visible. Por ejemplo, los comentarios pueden aparecer con un tipo de letra especial, y las proposiciones pueden aparecer con una indentación proporcional a la profundidad de su anidamiento en la organización jerárquica de las proposiciones.
3. Verificadores estáticos: Un verificador estático lee un programa, lo analiza e intenta descubrir errores potenciales sin ejecutar el programa. La parte del análisis es similar a la que se encuentra en los compiladores de optimación. Así un verificador estático puede detectar si hay partes de un programa que nunca se podrán ejecutar o si cierta variable se usa antes de ser definida. Además, puede detectar errores de lógica como intentar utilizar una variable real como apuntador, empleando las técnicas de verificación de tipos.
4. Intérpretes: En lugar de producir un programa objeto como resultado de una traducción, un interprete realiza las operaciones que implica el programa fuente. Para una proposición de asignación, por ejemplo, un intérprete podría construir un árbol como el de la figura 5, y después efectuar las operaciones de los nodos conforme "recorre" el árbol. En la raíz descubriría que tiene que realizar una asignación, y llamaría a una rutina para evaluar la expresión de la derecha y después almacenaría el valor resultante en la localidad de memoria asignada con el identificador posición. En el hijo derecho de la raíz, la rutina descubriría que tiene que calcular la suma de dos expresiones. Se llamaría a si misma de manera recursiva para calcular el valor de la variable inicial.

9. Diagrama de análisis de un programa fuente, definiendo cada una de sus partes

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).

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.

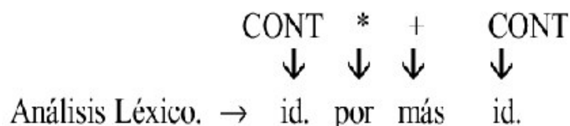


Figura 8.- Análisis léxico.

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 comoA/(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

Estructura del programa fuente
Programa fuente
Programa objeto en lenguaje ensamblador
Código de máquina relocalizable
Código de máquina absoluto
Figura 9.- Sistema para procesamiento de un lenguaje

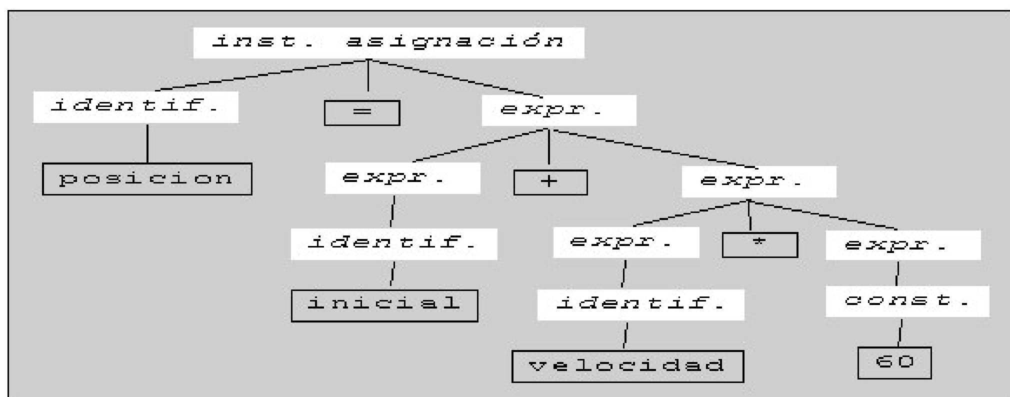


Figura 10.- Árbol de análisis sintáctico para posición := inicial + velocidad * 60.

10. Conclusiones

Este trabajo servirá mucho en el momento de la creación de un compilador, ya que en él se detallan todas y cada una de las partes que involucran a este. Primeramente investigue que existen distintos tipos de compiladores, me gustaria crear un compilador de optimación, ya que pienso que es muy útil a la hora de crear un algoritmo o programa. La función de un compiladores es leer un programa escrito es un lenguaje, en este caso el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto. Me parece fascinante que nosotros podamos crear un compilador en seis meses (en un curso), cuando en los años 50, ya que en aquellos tiempos se tardaron hasta 18 años trabajando en un compilador.
Por otro lado, comprendí que un compilador, requiere de una sintaxis y lenguajes específicos, ya que, al igual que el lenguaje humano, si no lo escribimos correctamente el compilador no hará lo que deseamos. Y que en la compilación hay dos partes: Análisis y Síntesis. La parte del análisis divide al programa fuente en sus elementos componentes y crea una representación intermedia. Aprendí que las herramientas que muestran tipos de análisis de programas fuente, son muy útiles al momento de crear un programa al codificar un algoritmo, ya que estas herramientas nos ayudan formateando el texto, corrigiendo errores, dando tips; para que nosotros como programadores seamos más eficientes al momento de crear alguna aplicación.
También he notado como todas nuestras materias se va complementando y enlazando, por ejemplo, en matemáticas discretas vimos la representación de árboles, los cuales usamos aquí. Igualmente en estructura de datos I, vimos métodos de ordenamiento que las gramáticas de los compiladores usan. Por lo tanto, no parece tan complicado crear un compilador, sólo se necesitan los conocimientos adecuados y dedicarle su tiempo para tener éxito.

11. Bibliografía

- Compiladores, Principios, técnicas y herramientas, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Addison – Wesley iberoamericana.
- http://www.dlsi.ua.es/docencia/asignaturas/comp1/comp1.html
- http://www.cps.unizar.es/~ezpeleta/COMPI
- http://www.ii.uam.es/~alfonsec
- Compiladores: Conceptos Fundamentales. B. Teufel, S. Schmidt, T. Teufel. Addison Wesley Iberoamericana.

Ceballos Carmona Miguel Ángel, 25 años
Estudiante universitario del ITESI, en Irapuato, Gto.

Trabajo para la materia Lenguajes y autómatas. Creado en Febrero del 2002.

Palabras clave para Búsqueda: compiladores, lenguajes y autómatas, análisis léxico, análisis

sintáctico, análisis semántico, árboles sintácticos, Compilador cruzado, Compilador con montador, autocompilador, metacompilador, descompilador, código objeto.

Publicado Originalmente en http://www.monografias.com/trabajos11/compil/compil2.shtml