Compiladores: un enfoque

Carlos Alberto Vanegas[*]

Resumen

En este artículo se expone un enfoque sencillo sobre compiladores, con la definición de la estructura de un compilador, donde se explican las fases de análisis y síntesis y cada una de las partes que componen dichas fases. También se expresa el funcionamiento del analizador lexicográfico y la función del análisis sintáctico. Por último, se muestran los posibles errores que se pueden producir en el proceso de compilación.

Palabras clave: lenguaje anfitrión, lenguaje embebido, análisis, síntesis, axiomas, *parse*, *token*, máquina virtual, autómata.

Abstract

In this articule a simple focus is described on compilers, defining the structure of a compiler, where the analysis phases and synthesis are explained and each one of the parts these phases are composed of. It is also expressed the lexicographical analyzer's operation and the function of the syntactic analysis. Lastly the possible errors that can take place in the compilation process are shown.

Words Keys: Language host, absorbed language, analysis, synthesis, axioms, parse, token, virtual machine, robot.

[*]Ingeniero de sistemas, especialista en Ingeniería de Software de la Universidad Distrital Francisco José de Caldas. Maestría de Ingeniería de Sistemas de la Universidad Nacional de Colombia. Docente de la Universidad Distrital, adscrito a la Facultad Tecnológica. Correo electrónico: cavanegas@udistrital.edu.co.

1. Introducción

1.1. Compilador: es un programa que recibe como entrada un programa escrito en un lenguaje de nivel medio o superior (el programa fuente) y lo transforma a su equivalente en lenguaje ensamblador (el programa objeto). La forma como llevará a cabo tal traducción es el objetivo central en el diseño de un compilador. Los traductores son clasificados en compiladores, ensambladores y preprocesadores.

1.2. Ensamblador: es el programa encargado de efectuar un proceso denominado de ensamble o ensamblado. Este proceso consiste en que, a partir de un programa escrito en lenguaje ensamblador, se produzca el correspondiente programa en lenguaje máquina (sin ejecutarlo), realizando:

- La integración de los diversos módulos que conforman al programa.
- La resolución de las direcciones de memoria designadas en el área de datos para el almacenamiento de variables, constantes y estructuras complejas; así como la determinación del tamaño de éstas.
- La resolución de los diversos llamados a los servicios o rutinas del sistema operativo, código dinámico y bibliotecas de tiempo de ejecución.
- La especificación de la cantidad de memoria destinada para las áreas de datos, código, pila y montículo necesaria, y otorgada para su ejecución.
- La incorporación de datos y código necesarios para la carga del programa y su ejecución.

1.3. Precompilador (preprocesador): programa que se ejecuta antes de invocar al compilador. Este programa es utilizado cuando el programa fuente, escrito en el lenguaje que el compilador es capaz de reconocer (*host language*), incluye estructuras, instrucciones o declaraciones escritas en otro lenguaje (*embeded language*).

1.4. Pseudocompilador: es un programa que actúa como un compilador, salvo que su producto no es ejecutable en ninguna máquina real, sino en una máquina virtual. Un pseudocompilador toma de entrada un programa escrito en un lenguaje determinado y lo transforma a una codificación especial llamada *código de byte*. Este código no tendría nada de especial o diferente al código máquina de cualquier microprocesador, si no fuera por el hecho de ser el código máquina de un microprocesador ficticio. Tal procesador no existe, en su lugar existe un programa que emula a dicho procesador, de ahí el nombre de máquina virtual. La ventaja de los pseudocompiladores es que permiten tener tantos emuladores como microprocesadores reales existan, pero sólo se requiere un compilador para producir un código que se ejecutará en todos estos emuladores.

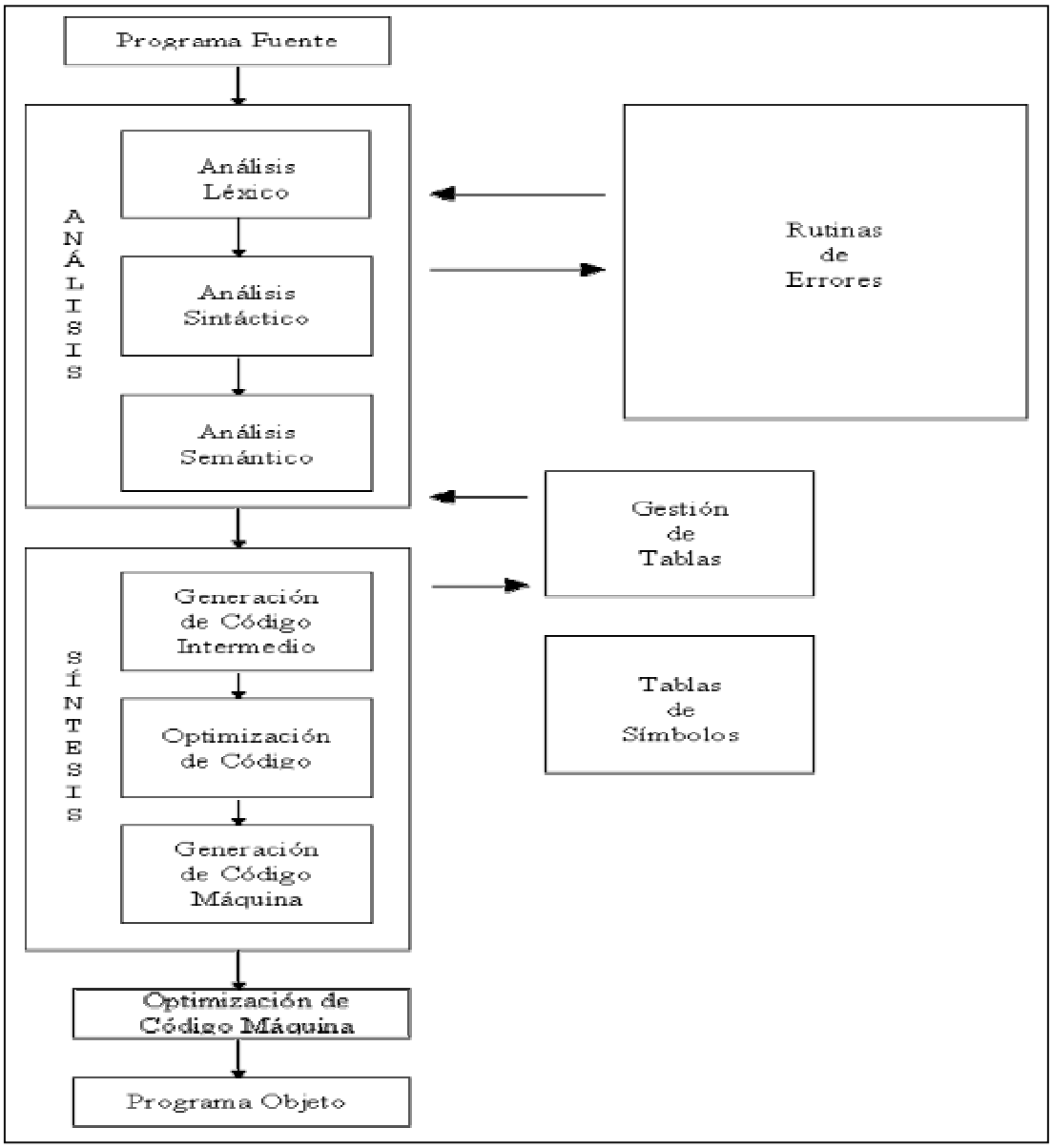
1.5. Intérprete: es un programa que ejecuta cada una de las instrucciones y declaraciones que encuentra conforme va analizando el programa que le ha sido dado de entrada (sin producir un programa objeto o ejecutable). La ejecución consiste en llamar a rutinas ya escritas en código máquina, cuyos resultados u operaciones están asociados de manera unívoca al significado de las instrucciones o declaraciones identificadas. Los intérpretes son útiles para el desarrollo de prototipos y pequeños programas para labores no previstas. Presentan la facilidad de probar el código casi de manera inmediata, sin tener que recurrir a la declaración previa de secciones de datos o código, y poder hallar errores de programación rápidamente.

Un conversor fuente a fuente traduce un lenguaje fuente de alto nivel a otro (por ejemplo, Java a C++). Una aplicación interesante de la traducción fuente-fuente es el desarrollo e implementación de prototipos de nuevos lenguajes de programación. Así, por ejemplo, si se desea definir un lenguaje especializado puede implementarse rápidamente mediante su traducción a un lenguaje convencional de alto nivel.

2. Estructura de un compilador

En un compilador se pueden distinguir dos fases principales: una fase de análisis, en la cual se lee el programa fuente y se estudia la estructura y el significado del mismo, y otra fase de síntesis, en la que se genera el programa objeto. Además, algunas estructuras de datos comunes, la más importante es la tabla de símbolos, junto con las funciones de gestión de ésta y una serie de rutinas auxiliares para detección de errores. (Véase Figura 1)

Figura 1. Esquema de un compilador



Fuente: http://borabora.univalle.edu.co/materias/compiladores/pl1.html

2.1. Fase de análisis

2.1.1. Analizador lexicográfico (explorador): es la parte del compilador que lee el programa fuente carácter a carácter, y construye, a partir de éste, unas entidades primarias llamadas *tokens*. Es decir, el analizador lexicográfico transforma el programa fuente en unidades lexicográficas. Las principales funciones que realiza son:

- Identificar los símbolos.
- Eliminar los blancos, caracteres de fin de línea, etc...
- Eliminar los comentarios que acompañan la fuente.
- Crear unos símbolos intermedios llamados *tokens*.
- Avisar de los errores que detecte.

Ejemplo: a partir de la sentencia en Java

nuevo:= viejo + cuenta * 2

genera un código simplificado para el análisis sintáctico posterior, por ejemplo:

<id1> <:=> <id2> <+> <id3> <*> <ent>

Nota: cada elemento encerrado entre <> representa un único *token*. Las abreviaturas *id* y *ent* significan identificador y entero, respectivamente.

2.1.2. Analizador sintáctico: comprueba que las sentencias que componen el texto fuente son correctas en el lenguaje, creando una representación interna que corresponde a la sentencia analizada. De esta manera, se garantiza que sólo serán procesadas las sentencias que pertenezcan al lenguaje fuente. Durante el análisis sintáctico, así como en las demás etapas, se van mostrando los errores que se encuentran.

Ejemplo: el esquema de la sentencia anterior corresponde al de una sentencia de asignación del lenguaje Java. Estas sentencias son de la forma:

<id> <:=> <EXPRESION>

y la parte que se denomina <EXPRESION> es de la forma:

<id> <+> <EXPRESION> o bien

<id> <*> <EXPRESION> o bien

<real>

La estructura de la sentencia queda, por tanto, de manifiesto mediante el siguiente esquema:

<id1><:=><EXPRESION> | <id2><+><EXPRESION> | <id3><*><EXPRESION> | <real>

2.1.3. Análisis semántico: se ocupa de analizar si la sentencia tiene algún significado. Se pueden encontrar sentencias que son sintácticamente correctas pero que no se pueden ejecutar porque carecen de sentido. En general, el análisis semántico se hace al mismo tiempo que el análisis sintáctico.

Ejemplo: en la sentencia que se ha analizado existe una variable entera. Sin embargo, las operaciones se realizan entre identificadores reales, por lo que hay dos alternativas: o emitir un mensaje de error "diferentes tipos de datos", o realizar una conversión automática al tipo superior, mediante una función auxiliar *floatValue()*.

<id1><=><EXPRESION>|<id2><+><EXPRESION>|<id3><*><EXPRESION>|
<real>|<floatValue>|<int>

2.2. Fase de síntesis

2.2.1. Generador de código intermedio: el código intermedio es un código abstracto independiente de la máquina para que se genere el código objeto. Éste ha de cumplir dos requisitos importantes: ser fácil de producir a partir del análisis sintáctico y ser fácil de traducir al lenguaje objeto. Esta fase puede no existir si se genera directamente código máquina, pero suele ser conveniente emplearla.

Ejemplo: consideremos un código intermedio de tercetos[1], la sentencia traducida a este código intermedio quedaría así:

temp1 = floatValue(2)

temp2 = id3 * temp1

temp3 = id2 + temp2

id1 = temp3

2.2.2. Optimizador de código: a partir de todo lo anterior crea un nuevo código más compacto y eficiente, eliminando, por ejemplo, sentencias que no se ejecutan nunca o simplificando expresiones aritméticas. La profundidad con que se realiza esta optimización varía mucho de unos compiladores a otros.

Ejemplo: para seguir con el ejemplo anterior, es posible evitar la función *floatValue()* mediante el cambio de 2 por 2.0, obviando además una de las operaciones anteriores. El código optimizado queda como sigue:

temp1= id3 * 2.0

id1 = id2 + temp1

2.2.3. Generador de código: a partir de los análisis anteriores, y de las tablas que estos análisis van creando durante su ejecución, se produce un código o lenguaje objeto que es directamente ejecutable por la máquina. Es la fase final del compilador. Las instrucciones del código intermedio se traducen una a una en código máquina.

Nota: cada instrucción de código intermedio puede dar lugar a más de una de código máquina.

2.3. Tabla de símbolos

Es el medio de almacenamiento de toda la información referente a las variables y objetos, en general, del programa que se está compilando.

[1] Llamado así porque en cada una de sus instrucciones aparecen como máximo tres operandos.

2.4. Rutinas de errores

Están incluidas en cada uno de los procesos de compilación (análisis lexicográfico, sintáctico y semántico), y se encargan de informar de los errores que encuentra en el texto fuente.

3. Funcionamiento de un analizador lexicográfico

Como se dijo anteriormente, el *analizador lexicográfico* lee el programa fuente, carácter a carácter, y construye, a partir de éste, unidades lexicográficas.

Ejemplo: sea la sentencia:

```
if (alfa < 718)
 alfa:= alfa + beta
```

El analizador lexicográfico daría como resultado la siguiente cadena de caracteres correspondiente a la misma:

```
(79);(12);(28);(13);(80);
 (12);(65);(12);(34);(12)
```

Como puede verse, el analizador lexicográfico ha simplificado el texto de entrada —consistente en una secuencia de símbolos— en otra cadena de caracteres, representados, cada uno de ellos, por un número, y que corresponden a los siguientes significados:

12 Variable real

13 Constante entera

28 Comparador <

34 Signo +

65 Asignador :=

Palabra reservada if

La información que da esta secuencia de caracteres es suficiente para el análisis de la estructura de la sentencia, pero no basta para un análisis de su significado, para lo cual hay que tener cierta información de cuáles son las variables que entran en juego en esta sentencia. Esto se soluciona mediante un segundo elemento que compone el *token*, que, por lo general, consiste en un puntero a la tabla de símbolos donde se hallan almacenadas las variables, con lo que queda, finalmente, una secuencia de pares:

(79,-);(12,32);(28,-);(13,7);(80,-);(12,32);

(65,-);(12,32);(34,-);(12,33)

En estos pares el primer elemento nos indica el tipo de objeto que estamos procesando según una tabla que nosotros hemos diseñado previamente (tabla de *tokens*). El segundo miembro de estos pares es, en caso necesario, un puntero a la tabla de símbolos o a la tabla de constantes.

El analizador lexicográfico, además, realiza ciertas tareas adicionales como:

- Identificar los símbolos.
- Crear unos símbolos intermedios llamados *tokens*.
- Eliminar comentarios del programa fuente.
- Eliminar los blancos, saltos de página, tabuladores, retornos de carro y demás caracteres propios del dispositivo de entrada.
- Reconocer las variables y asignarles una posición en la tabla de símbolos.
- Relacionar los mensajes de error que produce el compilador en sus diversas fases con la parte correspondiente del programa fuente (número de línea en que aparecen). En ciertos casos, el analizador lexicográfico se encarga también de producir el listado del programa con los errores de compilación.

Hay diversas razones por las que se separa la fase de análisis de un compilador en *análisis lexicográfico* y *análisis sintáctico*, éstas son las siguientes:

- En el diseño del analizador sintáctico, éste no ha de preocuparse de leer el archivo de entrada, ni de saltar blancos, ni comentarios, ni de recibir caracteres inesperados, puesto que todo ello ha sido filtrado previamente por el analizador lexicográfico.

- Se mejora la eficiencia del compilador en su conjunto. La lectura del programa fuente suele requerir gran parte del tiempo de compilación, que se ve reducido si el analizador lexicográfico incorpora técnicas especiales de lectura, o está realizado en ensamblador.

- Aumenta la portabilidad del compilador, ya que todas las diferencias que se produzcan en el alfabeto de entrada, o en el dispositivo de almacenamiento, pueden ser reducidas al analizador lexicográfico, dejando intacto al analizador sintáctico.

4. Función del análisis sintáctico

Analizar sintácticamente una cadena de *tokens* no es más que encontrar para ella el árbol sintáctico o de derivación que tiene como raíz el axioma de la gramática, y como nodos terminales la sucesión ordenada de símbolos que componen la cadena analizada. En caso de no existir este árbol sintáctico, la cadena no pertenecerá al lenguaje y el analizador sintáctico habrá de emitir el correspondiente mensaje de error. Existen dos formas de analizar sintácticamente una cadena:

- **Análisis descendente:** se parte del axioma inicial de la gramática y se va descendiendo utilizando las derivaciones izquierdas, hasta llegar a construir la cadena analizada.

- **Análisis ascendente:** se va construyendo el árbol desde sus nodos terminales. Es decir, se construye desde los símbolos de la cadena hasta llegar al axioma de la gramática.

Simultáneamente a la fase de análisis sintáctico, además de reconocer las secuencias de *tokens* y analizar su estructura, pueden realizarse una serie de tareas adicionales como:

- Recopilar información de los distintos *tokens* y almacenarla en la tabla de símbolos.

- Realizar algún tipo de análisis semántico, tal como la comprobación de tipos.

- Generar código intermedio.
- Avisar de los errores que se detecten.

5. Recuperación de errores lexicográficos

Los programas pueden contener diversos tipos de errores, que pueden ser:

- **Errores lexicográficos:** por ejemplo, lectura de un carácter que no pertenece al vocabulario terminal previsto para el autómata[2].
- **Errores sintácticos:** por ejemplo, una expresión aritmética con mayor número de paréntesis de apertura que de cierre.
- **Errores semánticos:** como la aplicación de un operador a un tipo de datos incompatible con el mismo.
- **Errores lógicos:** como un bucle sin final.

Cuando se detecta un error, un compilador puede detenerse en ese punto e informar al usuario, o bien, desechar una serie de caracteres del texto fuente y continuar con el análisis, y dar, al final, una lista completa de todos los errores detectados. En ciertas ocasiones es incluso posible que el compilador corrija el error, haciendo una interpretación coherente de los caracteres leídos. En estos casos, el compilador emite una advertencia, indicando la suposición que ha tomado, y continúa el proceso sin afectar a las sucesivas fases de compilación.

Los errores lexicográficos se producen cuando el analizador no es capaz de generar un *token* tras leer una determinada secuencia de caracteres. En general, puede decirse que los errores lexicográficos son a los lenguajes de programación, lo que las faltas de ortografía a los lenguajes naturales. Las siguientes situaciones producen, con frecuencia, la aparición de errores lexicográficos:

- Omisión de un carácter. Por ejemplo, si se ha escrito ELS en lugar de ELSE.

[2] Es un reconocedor de sentencias con un vocabulario T, cuya salida es el conjunto de dos valores que podemos denotar como *reconozco* y *no reconozco*. Cuando la máquina produce el valor *reconozco* significa que acepta la sentencia.

- Introducción de un nuevo carácter. Por ejemplo, si escribimos ELSSE en lugar de ELSE.
- Permutación de dos caracteres en el *token* analizado. Por ejemplo, si escribiéramos ESLE en lugar de ELSE.
- Un carácter ha sido cambiado. Por ejemplo, si se escribiera ELZE en vez de ELSE.

Las técnicas de recuperación de errores lexicográficos se basan, en general, en la obtención de los distintos sinónimos de una determinada cadena que hemos detectado como errónea. Por otra parte, el analizador sintáctico es capaz, en muchos casos, de avisar al analizador lexicográfico cuál es el *token* que espera que éste lea.

Todos los procedimientos para la recuperación de errores lexicográficos son, en la práctica, métodos específicos, y muy dependientes del lenguaje que se pretende compilar.

Conclusiones

- La función de un compilador es leer un programa escrito en un lenguaje, en este caso, el lenguaje fuente, y traducirlo a un programa equivalente en otro lenguaje, el lenguaje objeto.
- En el compilador existen dos fases: la de *análisis* y la de *síntesis*. En la fase de análisis, se lee el programa fuente y se estudia la estructura y el significado del mismo, y en la fase de síntesis se genera el programa objeto.
- Un compilador requiere de una sintaxis y de un lenguaje específico que permita la correcta traducción de un programa fuente a un programa objeto.
- Todo compilador debe tener implementado un detector de errores para facilitar la detección de errores lexicográficos, sintácticos, semánticos y lógicos.

Bibliografía

[1] Aho A, Sethi R, Ullman Jeffrey D. Compiladores: principios, técnicas y herramientas. Addisson-Wesley; 1986.

[2] Ceballos Carmona Miguel Ángel. Trabajo de lenguajes y autómatas. ITESI; 2002.

[3] Hopcroft John E, Ullman Jeffrey D. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley. USA: ISBN 0-201-02988-X; 1988.

[4] Teufel B, Schmidt S, Compiladores: conceptos fundamentales. Addisson-Wesley; 1993.

Infografía

http://www.dlsi.ua.es/docencia/asignaturas/comp1/comp1.html

http://www.cps.unizar.es/~ezpeleta/COMPI

http://www.ii.uam.es/~alfonsec

http://borabora.univalle.edu.co/materias/compiladores/pl1.html

http://homepage.mac.com/eravila/compiler.html