ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

"¿CÓMO DESARROLLAR Y PROBAR PROGRAMAS?, ¡COMPRUÉBALO!"

AUTORÍA
MARÍA CATALÁ CARBONERO

TEMÁTICA
PROGRAMACIÓN

ETAPA
CICLO MEDIO Y SUPERIOR DE INFORMÁTICA

Resumen

A la hora de realizar un programa en cualquier lenguaje de programación, no se puede entregar sin más, si no que éste debe pasar una serie de pruebas para comprobar que funciona correctamente, ya que es muy complicado, por no decir imposible, que un informático programe y el resultado sea un programa libre de errores y que funciona correctamente a la primera. Con la ayuda de compiladores, se puede conseguir que los programas funcionen correctamente antes de entregárselos al cliente.

Palabras clave

Programación, compilador, análisis semántico, análisis sintáctico, análisis léxico.

1. INTRODUCCIÓN

Cuando empezaron a desarrollar los ordenadores, el único lenguaje de programación disponible era el conjunto de instrucciones primitivas de cada máquina, conocido como lenguaje máquina.

Estos lenguajes máquina eran difíciles de leer y modificar por lo que se desarrollaron los lenguajes ensambladores, en los que se define un código nemotécnico para cada uno de las operaciones de la máquina, esto obliga a tener un conocimiento sobre las características físicas de la máquina.

Posteriormente aparecieron los primeros lenguajes de programación de alto nivel, que son independientes de la máquina pero que necesitan de un programa software (traductor) para que traduzca este lenguaje a un lenguaje entendible por la máquina.

A lo largo de este artículo vamos a ver el sistema o programa software encargado de traducir estos programas a lenguaje de alto nivel.

ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

2. COMPILADORES

Un compilador traduce un programa escrito en lenguaje de alto nivel o un lenguaje de cuarta generación, denominado programa fuente, a un programa escrito en lenguaje máquina o lenguaje ensamblador, denominado programa objeto.

2.1. Tipos de Compiladores

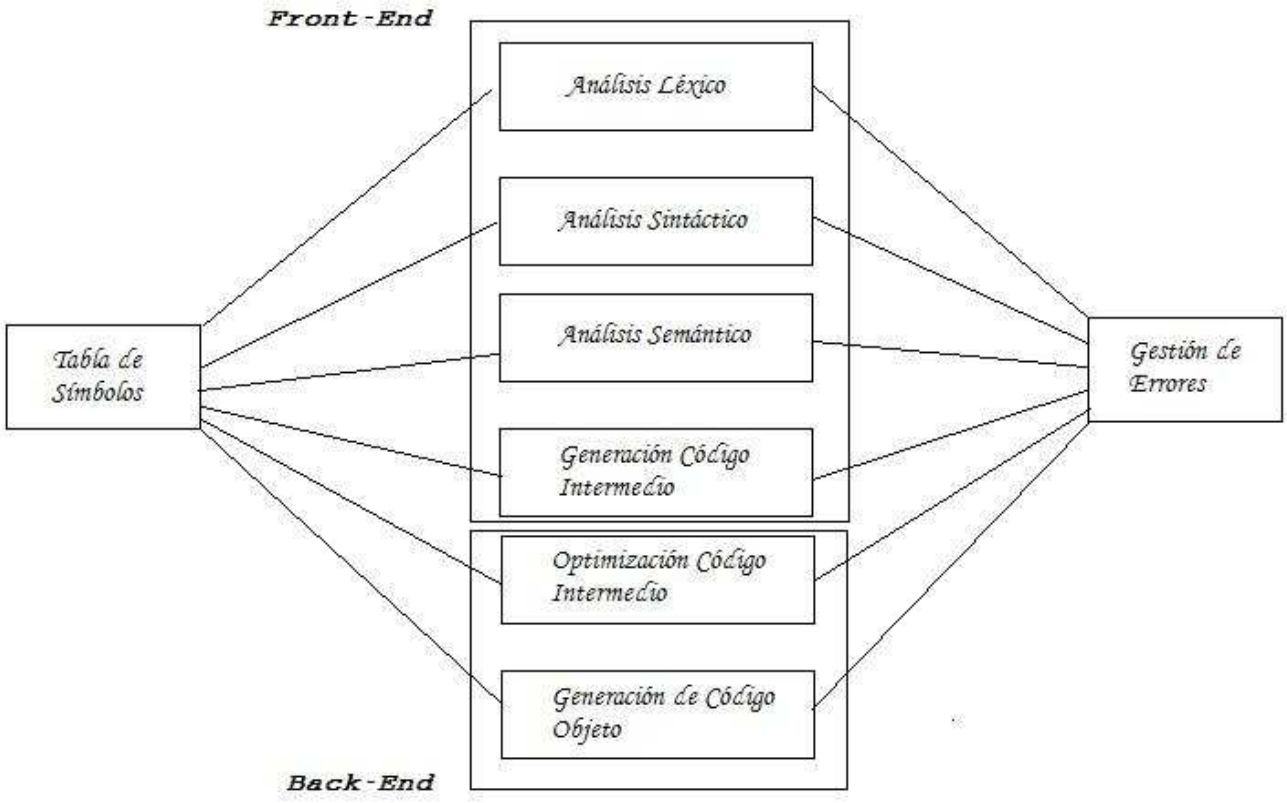
- *Ensamblador*: el lenguaje fuente es el lenguaje ensamblador.
- *Compilador cruzado*: genera código objeto para una máquina diferente de la que está utilizando para compilar.
- *Compilador con montador*: permite compilar varios módulos distintos y luego enlazarlos entre sí.
- *Autocompilador*: está escrito en el mismo lenguaje en el que se va a compilar.
- *Metacompilador*: compilador de compiladores, que a partir de la especificación formal de un compilador, lo genera.
- *Descompilador:* a partir de código en lenguaje máquina, trata de obtener código en un lenguaje de alto nivel.

2.2. Estructura de un compilador

- *Front- End*: es la parte que analiza el código fuente, comprueba su validez,etc.Implica la realización de un análisis de léxico, de la sintaxis y de la semántica.
- *Back*- End: es la parte que genera el código intermedio, su optimización y la generación de código final.

Además hay dos módulos que sirven de apoyo a todas las fases del compilador: *tabla de símbolos* y *Gestión de errores.*

ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009



2.3. Actividad sobre el Analizador Léxico

El Analizador léxico se encarga de leer el código fuente e ir traduciendo este a una secuencia de elementos básicos del lenguaje, denominado tokens. Este analizador aísla los tokens, identifica su tipo y los almacena en la tabla de símbolos.

Esta representación contiene la misma información que el programa fuente, pero en una forma más compacta, no estando el código ya como una secuencia de caracteres, sino de símbolos.

Ejercicio: Realizar el análisis léxico de la sentencia en lenguaje C:

```
Intensidad = (Magnitud + D ) * 17
```

ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

La forma más sencilla de resolver este problema sería el siguiente:

- Primero se extraen todos los token que se vayan encontrando en la expresión:

Identificador [1], Asignación, (, Identificador [2], Operador [+], Identificador [3],), Operador[*], Número [1].

- Segundo, se va rellenando el contenido de la tabla de símbolos:

	Referencia	***Nombre***
Identificadores	1	Intensidad
	2	Magnitud
	3	D
Constantes	1	17

2.4. Actividad sobre el Analizador Sintáctico

El Analizador Sintáctico recibe los tokens y comprueba su ordenación correcta y genera un árbol sintáctico que representa el programa.

La sintaxis de un lenguaje de programación especifica cómo deben escribirse los programas, mediante un conjunto de reglas de sintaxis o gramática del lenguaje. Un programa es sintácticamente correcto cuando sus estructuras aparecen en el orden correcto.

Así el analizador sintáctico, recibe la cadena de tokens procedentes del analizador léxico y busca en ella los posibles errores sintácticos que aparezcan

Para especificar este análisis se utiliza la gramática independiente del contexto, que está formada por los siguientes elementos:

- Un conjunto de símbolos no terminales { A, B, C…S}
- Un conjunto de símbolos terminales { a, b, c….}
- Un símbolo no terminal, llamado axioma S.
- Un conjunto de reglas de producción de la forma A:=a siendo “a” cualquier expresión en la que se utilicen tanto símbolos terminales como no terminales.

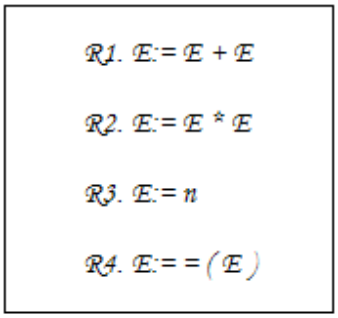
El análisis sintáctico va creando el árbol a partir del símbolo inicial y aplicando sucesivamente reglas de producción para obtener la representación abstracta de una secuencia de tokens.

ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

Ejercicio:

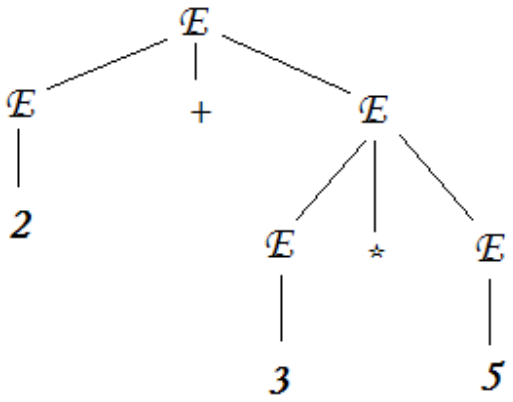
A partir de la expresión ″ 2 + 3 * 5 ″ crea al árbol sintáctico asociado a él.

- Se crean las reglas de producción:



Conjunto de reglas

- Se realiza el árbol sintáctico a partir de las reglas de producción:



2.5. Actividad sobre el Analizador Semántico

Este analizador comprueba que el árbol sintáctico es semánticamente válido, para ello genera un árbol semántico o etiquetado. Por lo tanto, en esta fase se revisa al programa fuente para tratar de encontrar errores semánticos, y reúne la información sobre los tipos para la fase posterior de generación de código.

ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

Un componente importante de este analizador es la verificación de tipos, en el que el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje. Además localiza los posibles errores de significado que aparezcan en el programa fuente, por ejemplo, intentar sumar número flotantes y asignarlos a un número entero.

Ejercicio:

Especifica la información que se almacenaría en la tabla de símbolos según el tipo de símbolo encontrado, así como el proceso interno que realiza el analizador semántico con dicha tabla.

- En primer lugar tendremos que analizar que tipo de símbolos nos encontramos y posteriormente especificar para cada tipo la información relevante. Un ejemplo de dicha información podría ser la siguiente:

Tipo de Símbolo	***Información***
Variable	Nombre, tipo, tamaño, dirección
Función	Nombre, tipo, comienzo del código
Tipo	Nombre, tipo, tamaño
Constante	Nombre, tipo, tamaño, valor

- En segundo lugar el proceso que se lleva a cabo es el siguiente:
  - Se van calculando los atributos
  - En función de los valores de estos se determina si el código fuente es semánticamente correcto
  - A la misma vez se van comprobando los tipos de las expresiones y rellenando la tabla de símbolos.
  - Una vez terminado éste, la tabla de símbolos estará rellena y el árbol de análisis semántico construido.

2.6. Actividad sobre la generación de Código Intermedio

El código intermedio se encarga de transformar un árbol semántico en una representación en un lenguaje intermedio cercano al código objeto.

ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

Si no se han producido errores en alguna de las etapas anteriores, este módulo realiza a partir de la tabla de símbolos y el árbol semántico, una traducción a un código interno propio del compilador, denominado código intermedio.

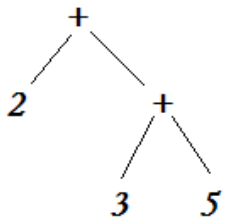
Para representar dicho código se pueden usar varias técnicas:

- *Código de tres direcciones:* cada instrucción contiene un código de operación, las direcciones de los operandos y la dirección donde se almacenará el resultado.
- *Árboles sintácticos abstractos*: el código se representa en forma de árbol donde cada nodo no terminal representa un operador y cada nodo terminal representa un operando.
- *Grafos dirigidos acíclicos*: surgen de los anteriores y en ellos se reutilizan expresiones comunes, siendo por tanto una representación más óptima.

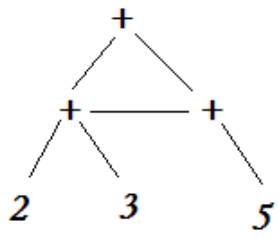
Ejercicio:

Representa la operación " 2 + 3 + 5 " a través de árboles sintácticos y la operación "(2 + 3) * (2 + 3 + 5)" a través de los grafos dirigidos.

- La representación con árboles sintácticos abstractos es la siguiente:



- Con grafos dirigidos acíclicos el código intermedio quedaría así:



ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

2.7. Actividad sobre el Optimizador de código

El optimizador de código realiza modificaciones sobre el código intermedio para mejorar la eficiencia en velocidad y tamaño.

Su misión consiste en recibir el código intermedio y optimizarlo atendiendo a determinados factores, tales como la velocidad de ejecución o el tamaño del programa objeto.

Estas optimizaciones pueden ser dependientes de la máquina o independientes, de estas últimas encontramos dos casos:

- *Factorización de expresiones*: Se optimizan las expresiones para que solo haya que calcularlas una vez.
- *Extracción de invariantes*: Expresiones que no cambian de valor durante la ejecución de un bucle, se extraen de éste para optimizar el resultado.

Ejercicio:

Optimiza las siguientes expresiones a través un método independiente de la máquina:

- **a)** A:= B + C + D

  E:= B + C + F

- **b)** REPEAT

  B:= 1

  A:= A – B

  UNTIL A = 0

En el apartado **a)** la optimización consiste en factorizar la parte común de las reglas de producción:

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

En el apartado **b)** se puede observar que existe un bucle, por lo que aplicaremos la extracción de invariantes:

ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

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

2.8. Generador de código final

El generador de código final transforma el código intermedio optimizado en código objeto de bajo nivel. En esta última etapa se genera el código objeto mediante la traducción del código intermedio optimizado.

2.9. Módulo de tratamiento de errores

Facilita la detección, y en algún caso, la recuperación de los errores producidos en las distintas fases de la compilación.

El compilador, cuando detecta un error:

- trata de buscar su localización exacta
- luego busaca su posible causa para presentar al programador un mensaje de diagnóstico, que será incluido en el listado de compilación.

2.10. Tabla de símbolos

Es la estructura que almacena toda la información relativa a constantes, variables, estructuras de datos y otros elementos pertenecientes al programa que se está compilando.

Esta información suele incluir:

- Tipo de cada elemento
- Sus dimensiones
- Y otras características

La tabla de símbolos está relacionada con todas las fases del proceso de compilación, y por tanto, es utilizada por cada uno de los módulos. Algunas de las fases se ocupan de completar esta tabla, mientras que otras utilizan la información en ella contenida.

ISSN 1988-6047 DEP. LEGAL: GR 2922/2007 Nº 21 – AGOSTO DE 2009

3. BIBLIOGRAFÍA

- Aho, y Ullman (1990). *Compiladores: Principios, técnicas y herramientas*. Madrid: Addison-Wesley Iberoamericana.
- Garrido A., y Iñesta J. (2002). *Diseño de Compiladores*. Universidad de Alicante.

Autoría

- María Catalá Carbonero
- IES Florencio Pintado, Peñarroya-Pueblonuevo, Córdoba
- mcata44@hotmail.com