TEMA 29

UTILIDADES PARA EL DESARROLLO Y PRUEBA DE PROGRAMAS. COMPILADORES. INTÉRPRETES. DEPURADORES.

ÍNDICE

1. INTRODUCCIÓN

Un **traductor** es un metaprograma que toma como entrada un programa (o parte de un programa) escrito en lenguaje simbólico, alejado de la máquina, denominado **programa fuente** y proporciona como salida otro programa, semánticamente equivalente, escrito en un lenguaje comprensible por el hardware del ordenador, denominado **programa objeto**. Veremos dos tipos de traductores, los compiladores e intérpretes, que representan dos aproximaciones muy distintas a la tarea de permitir el funcionamiento de los programas escritos en un determinado lenguaje de programación de alto nivel.

1.1. Notación polaca. Notación infija, sufija y prefija

Uno de los principales problemas que se presentaron al surgir los lenguajes de alto nivel fue saber cómo se genera el código máquina correspondiente a una expresión aritmética. Este problema, que está ya satisfactoriamente resuelto, es el que se estudia en este apartado.

Una **expresión** está formada por **operandos**, **operadores** y **delimitadores**. La misión de estos últimos es la de ayudar a conocer el orden en que se deben ejecutar los operadores. Por *ejemplo*, sea la expresión:

E := A * C - B * D + A / B ^ C

En la que no aparece ningún delimitador. Tal como está escrita puede tener varios significados y, por tanto, no es posible interpretarla.

Lo primero que hay que decidir es el orden en que se realizan las operaciones. Cada lenguaje definirá ese orden, aunque lo normal es que se realice de izquierda a derecha. Para especificar completamente el orden de evaluación de una expresión, se pueden utilizar paréntesis, es decir, delimitadores. La expresión anterior se podría formular:

E := ((A*C)-((B*D) + (A/(B ^ C))))

En la práctica, no se acostumbra a utilizar paréntesis más que cuando son necesarios. Dentro de cada paréntesis las operaciones se realizan según las prioridades de los operadores. Si, por ejemplo, se tiene A-B*C, se entiende que se quiere decir A-(B*C). En caso contrario, se escribiría (A-B)*C. Es decir, que si no se ponen paréntesis, la operación de multiplicar se realiza antes que la de sumar. En general, para poder interpretar una expresión que no está completamente parentizada, es necesario que cada operador tenga asignada una prioridad. En resumen, una expresión se evalúa de izquierda a derecha, realizándose siempre la operación de más prioridad. Cuando se quiere indicar un orden distinto se ponen delimitadores. Una posible tabla de prioridades para operadores aritméticos es la mostrada en la siguiente tabla:

Prioridad de operadores	
Operador	Prioridad
^	3
*, /	2
+, -	1

Con esta tabla ya puede evaluarse la expresión anterior. Si se quiere que el signo + tenga prioridad sobre el -, se escribirá:

E: = A * C - (B * D + A / B ^ C)

Para poder evaluar cualquier expresión hay que tener asignada prioridad a todos los operadores. En los lenguajes de programación aparecen tres tipos básicos de operadores: los aritméticos (+, -, *, / ...) los relacionales (<, >, = ...) y los lógicos (and, or, not), variando, de unos lenguajes a otros, el número de

operadores de cada grupo. Normalmente, los operadores lógicos son los de menor prioridad, siguen los relacionales y, por último, los aritméticos tienen la mayor prioridad.

El problema que se presenta es el de producir código objeto a partir de expresiones escritas en la forma anterior. Un simple análisis del problema muestra que para realizar la traducción directamente se necesita prever tal número de posibilidades que el algoritmo resultante necesita demasiado tiempo para realizar la traducción. La mejor solución se obtiene rescribiendo la expresión en otra notación, llamada **notación postfija** (o **sufija**). Las expresiones anteriores se han escrito en **notación infija**, que significa que el operador se coloca entre los operandos. En notación postfija (o **polaca inversa**) el operador se escribe detrás de los operandos. Por ejemplo:

A - B * C se escribe A B C * -

mientras que

(A - B) * C se escribe A B – C *

En notación postfija no se necesitan paréntesis ni prioridades. El orden de las operaciones queda determinado completamente por el orden en que se colocan los operadores. Se ejecuta primero el operador que se encuentra primero, recorriendo la expresión de izquierda a derecha.

Una tercera notación es la **prefija**, que consiste en colocar el operador delante de los operandos. Esta notación es exactamente inversa de la postfija, por lo que no presenta ninguna ventaja nueva para el problema que se considera. A esta notación también se le denomina **notación polaca**.

Ahora se verá cómo puede automatizarse la traducción a código objeto de expresiones escritas en notación postfija. Sea la expresión

A C * B D * A B C ^ / + -

correspondiente a la expresión puesta como ejemplo en notación infija. Para realizar la traducción se puede utilizar una pila en la que se van metiendo los operandos hasta que llega un operador. En ese momento, se sacan los operandos correspondientes y se inserta el operando resultado.

La traducción de expresiones en notación postfija es sumamente fácil. El problema estará resuelto si la traducción de expresiones de notación infija a notación postfija es también fácil.

En una expresión en notación infija los operandos están en el mismo orden que en la correspondiente expresión en notación postfija. Por tanto, lo único que varia es la colocación de los operadores. A través de las siguientes reglas se producen expresiones en notación postfija a partir de expresiones en notación infija:

a) Parentizar completamente la expresión.

b) Trasladar cada operador al lugar ocupado por su correspondiente paréntesis.

c) Suprimir todos los paréntesis.

Por *ejemplo*, para la expresión anterior con notación en infija:

E := ((A*C)-((B*D) + (A/(B ^ C))))

En notación postfija sería:

E := A C * B D * A B C ^ / + -

Este algoritmo de traducción de expresiones en notación infija a notación postfija es bueno para realizarlo normalmente. Sin embargo, si se quiere implementar en un compilador no resulta tan bueno, puesto que necesita recorrer la expresión dos veces: una para poner todos los paréntesis y otra para colocar los operadores. Un método mejor sería aquel que sólo necesite recorrer la expresión una vez, es

decir, que no necesite poner los paréntesis, sino que directamente vaya colocando los operadores en su sitio. Este método puede implementarse utilizando una pila en la que se van almacenando los operadores hasta sacarlos. Los operandos no se introducen en la pila, sino que, al colocarlos en las dos notaciones según el orden precedente, se situarán conforme se van leyendo.

2. COMPILADORES

Un **compilador** es un programa escrito en algún lenguaje; en general, en el lenguaje ensamblador de la correspondiente computadora, que admite como entrada un programa fuente y da como salida un programa objeto. La tendencia actual es la de escribir los compiladores en lenguaje de alto nivel para reducir los tiempos de programación y depuración. Incluso existen lenguajes diseñados especialmente para escribir compiladores.

El compilador traduce completamente un programa fuente, escrito en un lenguaje de alto nivel, a un programa objeto, escrito en lenguaje ensamblador o máquina. El programa fuente suele estar contenido en un fichero, y el programa objeto puede almacenarse como fichero en memoria masiva para ser procesado posteriormente, sin necesidad de volver a realizar la traducción. Una vez traducido el programa, su ejecución es independiente del compilador, así, por ejemplo, cualquier interacción con el usuario sólo estará controlada por el sistema operativo. Como parte importante de este proceso de traducción, el compilador informa al usuario de la presencia de errores en el programa fuente, pasándose a crear el programa objeto sólo en el caso de que no hayan sido detectados errores (por lo general, suele cancelarse la compilación al detectar un error).

2.1. Proceso de compilación

En esencia, la estructura lógica de un compilador se puede ver en la Figura. El proceso de compilación realiza primero un análisis del programa fuente para producir la sintaxis del programa objeto. Para ello utiliza diversas tablas.

Antes del proceso de compilación, se crea el programa fuente utilizando cualquier aplicación disponible con capacidades de edición de textos.

La traducción por un compilador (la **compilación**) consta de dos etapas fundamentales, que a veces no están claramente diferenciadas a lo largo del proceso: la etapa de análisis del programa fuente y la etapa de síntesis del programa objeto. Cada una de estas etapas conlleva la realización de varias fases. El análisis del texto fuente implica la realización de un análisis del léxico, de la sintaxis y de la semántica. La síntesis del programa objeto conduce a la generación de código y su optimización.

Para realizar estas funciones las estructuras deben obedecer a reglas muy rígidas, puesto que las ambigüedades no podrían ser dilucidadas por la máquina. De ahí la importancia de los lenguajes formales y de los autómatas en relación con los lenguajes evolucionados y los compiladores, pues las tareas esenciales del compilador son la evaluación sintáctica y la evaluación semántica (o traducción propiamente dicha).

Para llevar a cabo la compilación, el programa debe residir en memoria principal simultáneamente con el compilador. El resultado de la compilación puede dar lugar a la aparición de errores, en cuyo caso no se genera el programa objeto, sino que se realiza un **listado de compilación**, un informe indicando la naturaleza y situación de los errores detectados por el compilador en el programa fuente. Con el programa editor se corrigen y se comienza de nuevo el proceso.

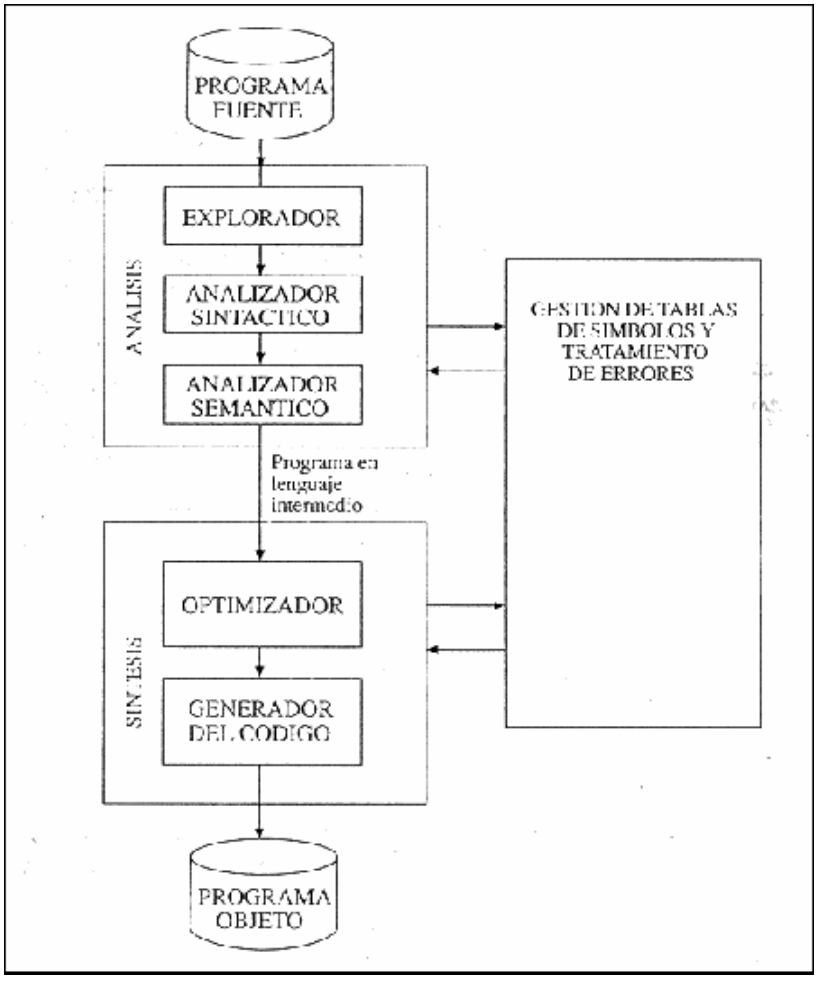
Una vez finalizada la compilación y obtenido el programa objeto, se somete a un proceso de montaje donde se enlazan los distintos módulos que lo componen, en caso de tratar con programas que poseen subprogramas (que pueden ser compilados separadamente). Además, se incorporan las denominadas **rutinas de biblioteca** en caso de solicitarlas el propio programa. Este proceso de montaje es realizado por un programa denominado **montador** o **encuadernador**, que también recibe el nombre de *editor de enlace* o *linker*.

La compilación es un proceso complejo que consume a veces un tiempo muy superior a la propia ejecución del programa. Este proceso consta, en general, de dos etapas fundamentales: la etapa de **análisis** del programa fuente y la etapa de **síntesis** del programa objeto. Cada una de estas etapas conlleva la realización de varias fases.

El compilador utiliza internamente una **tabla de símbolos** para introducir determinados datos que necesita. Esta tabla interviene prácticamente en todas las fases del proceso de compilación; así mismo, el compilador posee un **módulo de tratamiento de errores** que permite determinarlas reacciones que se deben producir ante la aparición de cualquier tipo de error.

El **explorador** (scanner) o analizador lexicográfico tiene como misión indagar sobre los caracteres del programa fuente y generar los símbolos del programa para pasarlos a los analizadores sintáctico y semántico.

Por último, se realiza la **generación del código**, que es la traducción del programa fuente interno a lenguaje máquina o a un lenguaje intermedio.



3. ANÁLISIS DEL PROGRAMA FUENTE

Durante esta etapa, el programa fuente se divide en sus elementos componentes, creándose una representación intermedia del mismo. Durante este proceso, es posible detectar posibles errores en la escritura del programa.

3.1. Análisis lexicográfico

Se examina el programa fuente de izquierda a derecha, reconociendo las unidades básicas de información pertenecientes al lenguaje. Estas unidades básicas se denominan **unidades léxicas** o *tokens.* Un **token** es un elemento o cadena (secuencia de caracteres) que tiene un significado propio en el

lenguaje. Además de reconocer cada token, almacena en la tabla de símbolos aquella información del símbolo que pueda ser necesaria para las restantes fases de compilación.

Ejemplo:

A * B + C

A ---- Es un identificador.

* ---- Es un operador.

Normalmente, en un lenguaje se tienen los siguientes grupos de caracteres:

- Constantes.
- Identificadores.
- Operadores (aritméticos, relacionales, lógicos, etc.).
- Palabras clave (if, goto, while, etc.)
- Delimitadores ([, ;, (, etc.)

Cada uno de los grupos anteriores forma un token (unidad léxica) distinta; y a cada token se le asocia una tabla donde se reflejan todos los caracteres pertenecientes al mismo.

El léxico de un lenguaje es el conjunto de tokens distintos que lo forman.

Al conjunto de reglas que permite escribir correctamente un lenguaje se le llama **sintaxis del lenguaje**.

El programa fuente se transforma en una tabla. Cada elemento de la tabla representa un carácter del programa fuente y se le llama **descriptor**. Estos tienen todos el mismo formato y contienen un código descriptor (que identifica a qué tokens pertenece) y un código de dirección (que indica la posición dentro del token).

Por *ejemplo*:

FOR i := 1 TO n DO A := B * C;

El escáner (programa que realiza el análisis lexicográfico) trocea la secuencia en unidades que tienen sentido:

FOR --- (palabra clave, 35)

donde 35 es la posición de FOR en la tabla de las palabras claves. i

--- (identificador, 50)

donde 50 es la posición de i en una tabla de identificadores.

:= --- (delimitador, 20)

donde 20 es la posición de := en una tabla de delimitadores. Y así sucesivamente.

La técnica del reconocimiento de la serie de caracteres del programa fuente se basa en la utilización de algoritmos con el principio del autómata finito. El autómata que se suele utilizar para el análisis lexicográfico tiene dos tipos de paradas:

V (verdadera, que indica que la serie examinada es correcta) y

F (falsa, que indica que la serie no es correcta).

En el caso de que no existan errores en este primer análisis, se obtiene una representación del programa formada por:

- La descripción de símbolos en la tabla de símbolos.
- Una secuencia de símbolos en la que se incluye, junto a cada símbolo, una referencia a la ubicación de dicho símbolo en la tabla de símbolos. Esta secuencia es denominada tira de tokens.

En esta nueva representación, se ha eliminado previamente toda la información superflua, como comentarios, tabulaciones y espacios en blanco no significativos.

Los errores que pueden ser reconocidos se deben a la detección de cadenas de caracteres del programa fuente que no se ajustan al patrón de ningún token, como identificadores de variables incorrectos, números expresados incorrectamente, cadenas de caracteres mal delimitadas, etc.

3.2. Análisis sintáctico

La sintaxis de un lenguaje de programación especifica cómo deben escribirse los programas basándose en un conjunto de reglas sintácticas que determinan la gramática del lenguaje. Un programa es sintácticamente correcto cuando sus estructuras (expresiones, sentencias, asignaciones, etc.) aparecen dispuestas de forma correcta de acuerdo a la gramática del lenguaje.

El análisis sintáctico recibe la tira de tokens del analizador lexicográfico e investiga en ella los posibles errores sintácticos que aparezcan, como expresiones estructuradas incorrectamente, falta de algún token dentro de una sentencia, formato incorrecto en una asignación, etc.

En caso de no encontrar errores, se agrupan los componentes léxicos en frases gramaticales que serán utilizadas en la síntesis del programa objeto. Estas frases gramaticales se suelen representar mediante un árbol sintáctico que pone de manifiesto la estructura jerárquica de los componentes de un programa.

El papel del analizador sintáctico de un compilador es tomar la cadena producida por el explorador y, utilizando un algoritmo analizador o reconocedor (parser), determinar si es o no una cadena válida, es decir, si es sintácticamente correcta de acuerdo con la gramática del lenguaje utilizado.

Además, mediante una estructura arbórea se produce una salida que facilita la labor de la fase siguiente.

Últimamente se han desarrollado muchos algoritmos analizadores, pero sólo unos pocos son usados en los compiladores que se escriben actualmente. Se pueden clasificar en **reconocedores** descendentes y reconocedores ascendentes. De ellos se hablará en otro apartado.

3.3. Análisis semántico

Una construcción de un lenguaje de programación puede ser sintácticamente correcta y carecer de significado. Por tanto, si queremos generar código en lenguaje máquina con el mismo significado que el código fuente, tendremos que determinar la validez semántica de las construcciones del código fuente.

El análisis semántico se encarga de estudiar la coherencia semántica del código fuente a partir de la identificación de las construcciones sintácticas y de la información almacenada en la tabla de símbolos. El análisis semántico debe ser capaz de detectar construcciones *sin un significado correcto*. Por ejemplo, asignar a una variable de tipo entero un valor de tipo cadena de caracteres, sumar dos valores booleanos, etc.

Como la gramática de contexto libre no llega a definir completamente al lenguaje fuente, se complementa ésta con las **rutinas semánticas**, que tienen una doble misión:

- Completar la definición del lenguaje fuente.
- Realizar la traducción del lenguaje fuente al LI.

4. SÍNTESIS DEL PROGRAMA OBJETO

Construye el programa objeto a partir de la representación obtenida en la etapa de análisis.

4.1. Generación del código intermedio

Se traduce el resultado de la etapa de análisis (en el caso de ausencia de errores) a un **lenguaje intermedio** propio del compilador. Esta representación intermedia (código intermedio) debe poseer las siguientes características:

- Debe ser independiente de la máquina.
- Debe ser fácil de producir.
- Debe ser fácil de traducir a un lenguaje máquina.

Esta fase aparece, generalmente, en aquellos compiladores construidos para su implantación en distintas máquinas, ya que su existencia hace que todos los módulos de análisis del compilador sean comunes para todas las máquinas (aumenta la portabilidad del compilador y se abaratan sus costes de construcción).

Cabría considerar la compilación en dos pasos: primero, de Lenguaje Fuente (LF) a Lenguaje Intermedio (LI) y, luego, de LI a Lenguaje Objeto (LO).

Cuando la disparidad entre el LF y el LO es muy elevada, surge la conveniencia de diseñar y usar un LI que facilite el proceso.

Las ventajas mayores del empleo de un LI son las siguientes:

- Facilitar la fase de optimización de un compilador.
- Aumentar la portabilidad del compilador de una máquina a otra.
- Facilitar la división del proyecto en fases.

Problema de la portabilidad. Supóngase que se tiene un LI que permite representar a p LF distintos. Además, supóngase que cada uno de estos p lenguajes se quieren introducir en q máquinas diferentes. Pues bien, si no se tuviera un LI, se necesitarían p*q compiladores diferentes, uno para cada lenguaje y para cada máquina y, aunque se pudieran aprovechar módulos, serían proyectos distintos. Pero con un LI solo harían falta p + q compiladores; p para pasar cada lenguaje fuente al LI y luego q, que transformen el LI en el lenguaje objeto de cada máquina o computadora.

4.2. Optimización de código

Esta fase se suele introducir para mejorar la eficiencia del código producido por el compilador. Se toma el código intermedio de forma global y se optimiza, adaptándolo a las características del procesador al que va dirigido para mejorar los resultados de su ejecución.

La optimización se realiza en cualquier parte del programa, pero es en los bucles donde se consiguen los mejores resultados; por ejemplo, sacar una sentencia que se repite fuera del bucle, consiguiendo que sólo se ejecute una vez. De esta forma, se consigue un programa igualmente correcto, pero que consume menos tiempo de ejecución.

Un problema de la optimización es que incrementa el tamaño y la complejidad del compilador.

La correspondencia entre código fuente y código objeto ejecutable nunca es biunívoco; no solamente por el hecho de las diferentes modalidades de generación de dicho código, sino -además y fundamentalmente- por las posibles optimizaciones que pudieran surgir.

Pueden distinguirse dos clases de optimizaciones susceptibles de ser tratadas por una fase compiladora:

1. Optimizaciones independientes de la máquina.
2. Optimizaciones dependientes de la máquina.

Optimizaciones independientes de la máquina

Existen multitud de tipos y subtipos de esta clase. Algunos de ellos son: la reducción simple, el reacondicionamiento de instrucciones, la reducción de potencia y la reducción de frecuencias.

a) Reducción simple

Si la instrucción original es:

```
A := 4 * 5 - 3;
```

La optimización debería dejar la instrucción como:

```
A := 17;
```

b) Reacondicionamiento de instrucciones

De forma que se evite en lo posible la utilización excesiva de variables temporales.

c) Reducción de potencia

Este tipo de optimizaciones consiste en reemplazar una operación considerada como compleja por otra de más fácil realización y que conlleve menos tiempo de ejecución o menos espacio. La siguiente instrucción:

```
A := J**2;
```

se puede transformar en

```
A := J*J;
```

con ello se mejora en tiempo de ejecución, dado que la operación de multiplicación suele estar implementada habitualmente por procedimientos hardware.

d) Reducción de frecuencias

La repetición de una serie de instrucciones, innecesaria para la lógica del programa, es una circunstancia bastante frecuente, sobre todo en usuarios poco habituados en la programación. Un buen optimizador de código deberá desplazar este tipo de instrucciones hacia lugares del programa en donde su ejecución sea lo menos reiterada posible.

Optimizaciones dependientes de la máquina

La asignación de registros para cálculos y la optimización particular de expresiones aritméticas son ejemplos clásicos de este tipo de circunstancias absolutamente dependientes de la computadora.

Supóngase una máquina general M que dispone de n registros de uso general numerados del R_1, al R_n. Se desea, en este momento inicializar un registro general cualquiera R_k con un valor predeterminado V. Las circunstancias u opciones que pueden presentarse en este caso son las siguientes:

1. El valor V ya se encuentra cargado en el registro R_k. El problema, por consiguiente, está resuelto.

2. El valor V no se encuentra en ninguno de los n registros generales, pero se da la circunstancia de que el registro R, está libre. Bastará, pues, con ejecutar una instrucción de la forma

```
LDA Rt, V
```

3. El valor V no se encuentra en ninguno de los registros generales y, además, ninguno de ellos está libre. Debe, por consiguiente, salvarse un registro para proceder a la carga del valor V; pero, ¿cuál de

ellos? En general, podrá utilizarse aquel registro cuyo número de referencias sea menor para la propia instrucción en curso (algoritmo LRU).

Es fácilmente demostrable que la asignación óptima de registros generales es tanto mejor cuanto menor es el número de LDA y STA que se necesitan.

4.3. Generación de código objeto

Se traduce el código intermedio optimizado a código final, es decir, al lenguaje máquina (en algunos casos a ensamblador) del procesador al que va dirigido el compilador. Para ello, se reserva la memoria necesaria para cada elemento del código intermedio, y cada instrucción en lenguaje intermedio es traducida a varias instrucciones en lenguaje máquina.

El *algoritmo de generación de código*, partiendo de una sentencia en polaca inversa, es análogo al de su evaluación; sólo que, en vez de evaluar el operador con sus operandos, se debe generar código que permita esta misma evaluación cuando se ejecute el programa objeto. El algoritmo consta de los dos pasos siguientes:

1. Se va analizando secuencialmente cada símbolo de entrada. Si es operando, se apila y sigue con este mismo paso, pero con el símbolo siguiente, etc. Si el símbolo de entrada es un operador, se va al paso 2.

2. Si el símbolo en cuestión es un operador (por ejemplo, binario), se aplica a los dos símbolos superiores de la pila. Así, por ejemplo, si se tenía en la pila A B, los dos elementos superiores, y llega el operador *, entonces se debe generar LOAD A, seguido de MULT B y de STORE T.

5. DETECCIÓN Y TRATAMIENTO DE ERRORES

Aunque es en la etapa de análisis donde se detectan la mayoría de los errores, cada etapa del proceso puede encontrar errores. Para tratar adecuadamente un error, interviene el **módulo de tratamiento de errores** que comprende generalmente la realización de dos acciones:

- *Diagnóstico del error*: Trata de buscar su localización exacta y la posible causa del mismo, para ofrecer al programador un mensaje de diagnóstico, que será incluido en el listado de compilación.
- *Recuperación del error*: Después de detectado un error, cada fase debe tratarlo de alguna forma para poder continuar con la compilación y permitir la detección de más errores, aunque no se genere código objeto.

Los **tipos de errores** que puede tener un programa son los siguientes:

- *Errores lexicográficos*: Se producen por la aparición de tokens no reconocibles, es decir, cadenas que no se ajustan al patrón de ningún símbolo elemental del lenguaje.
- *Errores sintácticos*: Aparecen cuando se violan las reglas de sintaxis del lenguaje.
- *Errores semánticos*: Se detectan cuando aparece una secuencia sintácticamente correcta, pero que carece de sentido dentro del contexto del programa fuente.
- *Errores lógicos*: Son debidos a la utilización de un algoritmo o expresión incorrecta para el problema que se trata de resolver.
- *Errores de ejecución*: Son errores relacionados con desbordamientos, operaciones matemáticamente irresolubles, etc.

En ocasiones, se indican determinados errores que pueden existir pero que no perjudican al resto del proceso de compilación e incluso pueden permitir el funcionamiento del programa final. Estos mensajes de error se denominan **advertencias** o *warnings*. Un ejemplo típico puede ser la declaración de una variable que no se utiliza en ningún bloque del programa.

6. GRAMÁTICAS FORMALES

Alfabeto A es un conjunto de símbolos llamados **símbolos terminales**.

Una **fórmula**, cadena o sentencia es una concatenación de símbolos de un alfabeto.

Por **A'** se designa el conjunto de todas las posibles cadenas que se pueden formar con el alfabeto A.

Lenguaje L es un subconjunto del conjunto A'.

Símbolos terminales son los símbolos del alfabeto A, y los **símbolos no terminales** son un conjunto de símbolos, no pertenecientes a A, que representan estados intermedios en el proceso de generación. El **símbolo de arranque** es un símbolo no terminal distinguido, desde el que se generan todas las sentencias del lenguaje.

Regla de producción es una regla de transformación de cadenas. Consta de una parte izquierda y una derecha. La parte izquierda representa una cadena que contiene la subcadena que va a transformarse. La parte derecha nos da la cadena resultante de la transformación.

Gramática formal G es una cuádrupla G = (Et, Ea, P, S).

Et	Alfabeto de símbolos terminales.
Ea	Alfabeto de símbolos no terminales o variables.
P	Reglas de escritura o producciones.
S	Símbolo distinguido o inicial.

Ejemplo:

Una gramática de tipo 3 es aquella que tiene estas reglas de producción:

1. Partiendo del estado A se puede producir la secuencia aB, es decir:

   A aB

2. Partiendo del estado A se puede producir la secuencia a, es decir:

   A a

En esta gramática *A* y *B* pertenecen al alfabeto de variables o símbolos no terminales y *a* pertenece al alfabeto de símbolos terminales. El símbolo inicial en este caso es *A*.

Expresiones de las gramáticas

Forma normal de Backus (BNF)

Este apartado va a mostrar cómo se puede describir un lenguaje -la gramática de ese lenguaje y, en particular, las leyes de deducción-, es decir, la sintaxis. La descripción de un lenguaje se efectúa mediante un **metalenguaje**, que tiene que ser distinguible del lenguaje para evitar paradojas, como sería el caso de querer describir el castellano mediante el propio castellano.

Un metalenguaje para definir lenguajes de programación que está muy extendido es el **BNF**, el cual inicialmente se desarrolló para describir el lenguaje Algol.

En el metalenguaje BNF los símbolos terminales del lenguaje descrito se representan por ellos mismos.

Los símbolos no terminales se representan mediante los metaparéntesis < >.

La flecha de las leyes se representa por el símbolo ::=.

Si hay varias leyes con la misma parte izquierda, se pueden representar en una sola separando las diversas partes derechas por el símbolo metalingüístico |, que significa "O exclusivo".

Este sistema facilita que los símbolos de los alfabetos sean formados por varios símbolos; así se puede definir en Algol:

<dígito> ::= 0|1|2|3|4|5|6|7|8|9

en la que se emplea "dígito" en vez de un solo símbolo no terminal y se agrupan en una sola definición diez leyes de deducción.

Cuando se utilizan las llaves, éstas indican que lo que encierran se puede repetir cero o un número arbitrario de veces, es decir, se trata de la definición:

<Lista de parámetros> ::= <parámetro> | <Lista de parámetros>, <parámetro>

que se transforma, utilizando llaves, en la siguiente:

<lista de parámetros> ::= <parámetro> [,<parámetro>]

Diagramas sintácticos

Tienen la ventaja de ser gráficos y verse mejor que el BNF. Es posible agrupar varias reglas de formación en un único diagrama. Con estos diagramas se simplifican mucho las reglas de formación.

7. PROGRAMAS OBJETO

El programa objeto se llama **absoluto**, si debe cargarse en unas posiciones fijas de memoria, y **localizable** (relocatable o reubicable), si se puede cargar en cualquier lugar de la memoria.

Además, antes de cargar el programa objeto, es necesario unirlo a otros programas objeto (subrutinas de E/S, subprogramas de funciones, etc.).

Los formatos de los programas objeto localizables son casi tan flexibles como los utilizados para los lenguajes de nivel ensamblador, aunque el montaje en memoria requiere bastante tiempo. Esta función la realiza el montador de enlaces; mientras que, en los sistemas operativos complejos, la carga final la efectúa el cargador. Si el sistema operativo es más sencillo, existe un solo paso realizado por el cargador enlazador.

El programa objeto consta de cuatro partes: el diccionario de símbolos externos, el texto, el diccionario localizable y la ficha END.

- El *diccionario de símbolos externos* define las secciones de control, las referencias externas, los puntos de entradas y las áreas COMMON.
- El *texto* comprende las instrucciones en lenguaje de máquina y los datos del programa objeto.
- El *diccionario localizable* contiene las direcciones que tienen que modificarse en el momento de la ejecución.
- La *ficha END* indica la terminación del programa objeto.

Estos formatos pueden variar según el tipo de sistema operativo.

8. MÉTODOS DE RECONOCIMIENTO

El análisis sintáctico comprueba si la sintaxis es correcta o no y si la cadena pertenece al lenguaje o no. Existen dos formas básicas de realizar el análisis:

a) **Reconocimiento descendente** (top-down). Parte del símbolo distinguido de la gramática y llega a la sentencia que se quiere reconocer mediante las reglas de producción de la gramática.

b) **Reconocimiento ascendente** (bottom-up). Parte de la sentencia que se quiere reconocer y tiene que llegar al símbolo distinguido de la gramática.

Dentro de cada reconocimiento existen diferentes tipos que se diferencian en la forma de aplicar las reglas de producción. Otros tipos de reconocimiento son:

- *Reconocimiento descendente recursivo*. Es el más antiguo, poco potente y propenso a cometer errores.

- *Reconocimiento ascendente basado en la gramática del operador*. Tiene los mismos problemas que el anterior.

Estos tipos de reconocimiento están superados por dos tipos de reconocedores más potentes, de implementación automática y más estudiados hoy en día. La gramática que pueden utilizar es más amplia, pues no existen tantas restricciones. Estos **reconocedores** son los **LL** y **LR**.

8.1. Reconocimiento descendente

El problema radica en saber en cada momento qué regla se puede aplicar o se debe aplicar. En principio puede hacerse por un método de prueba y error.

Supóngase la siguiente gramática:

S cAd
A ab | a

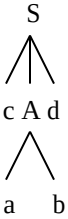
para reconocer la cadena:

W = c a d

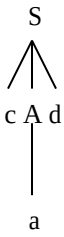
la forma de actuación sería:

1. Mira de izquierda a derecha (también lo podría hacer al contrario) y busca la regla de producción que empiece por el símbolo distinguido.

2. Se busca el primer símbolo no terminal. En este caso será A. Se busca una regla de producción para A que lleve a “a”.



Hay error, pues no es la sentencia que hay que reconocer. Se vuelve atrás y en un segundo intento se tendrá la sentencia que se quería reconocer.



Esto es lo que se conoce con el nombre de reconocedor descendente con vuelta atrás.

Los problemas que surgen son los siguientes:

1. Recursión por la izquierda. Traería como consecuencia una recursión ilimitada por la izquierda.

   A Ab

Una forma de evitarlo es crear un símbolo terminal nuevo.

2. Se pueden rechazar sentencias que están bien.

3. Se pueden aceptar sentencias que están mal.

4. No se puede especificar bien dónde se ha producido el error, debido a la vuelta atrás.

8.2. Reconocimiento ascendente

Los reconocedores ascendentes parten de las sentencias e intentan -una vez utilizadas las reglas de producción- llegar al símbolo distinguido de la gramática. Si se llega a dicho símbolo, la sentencia es aceptada.

El problema es más complejo que en el planteamiento de reconocimiento descendente. Ahora hay que buscar una parte derecha que va a tener símbolos terminales o no. En general, se tienen más posibilidades que en el caso descendente.

Para realizar este reconocimiento se necesita una pila y una serie de operaciones, que se van a aplicar dependiendo del estado en que se encuentre el autómata.

Descripción de las operaciones:

a) *Desplazar*: Consiste en pasar a la pila el símbolo que toca leer en la entrada.

b) *Reducir*: Elegida una serie de símbolos del tope de la pila, se sustituye por el símbolo terminal que corresponda según las reglas de producción.

c) *Aceptar*: Cuando se llega al final del proceso y la entrada está vacía, en la pila sólo se dispone de un elemento llamado “símbolo distinguido”.

d) *Error*: Cuando la sentencia no pertenezca a la gramática que se está aplicando, es decir, cuando la entrada esté vacía y no se tenga el símbolo distinguido.

9. INTÉRPRETES

Un **intérprete** permite que un programa fuente escrito en un determinado lenguaje vaya traduciéndose y ejecutándose directamente, sentencia a sentencia, por el ordenador. El intérprete capta una sentencia fuente, la analiza e interpreta, dando lugar a su ejecución inmediata, no creándose, por tanto, un archivo o programa objeto almacenaje en memoria masiva para posteriores ejecuciones. La ejecución del programa estará supervisada por el intérprete.

En la práctica, el usuario crea un archivo con el programa fuente. Esto suele realizarse con un editor específico del propio intérprete del lenguaje. Según se van almacenado las instrucciones simbólicas, se analizan y se producen los mensajes de error correspondientes; así, el usuario puede proceder inmediatamente a su corrección. Una vez creado el archivo fuente, el usuario puede dar la orden de ejecución y el intérprete lo ejecuta línea a línea. Siempre el análisis antecede inmediatamente a la ejecución.

Si utilizamos un intérprete para traducir un programa, cada vez que necesitemos ejecutar el programa se volverá a analizar, ya que, no se genera un fichero objeto. En cambio, con un compilador, aunque sea más lenta, la traducción sólo debe realizarse una vez. Además los traductores no permiten realizar optimizaciones del código (que eliminan órdenes innecesarias compactando el código) más allá del contexto de cada sentencia del programa.

En general, los intérpretes se componen de dos partes fundamentales:

a. Conjunto de subprogramas que corresponden al juego de instrucciones del lenguaje simbólico.

b. Un bucle de interpretación que realiza las funciones de codificación de una instrucción y la determinación de los operandos así como las operaciones correspondientes a cada subprograma.

Diferencias entre compiladores e intérpretes

Claramente se puede deducir que las diferencias entre compiladores e interpretes radican únicamente en la forma de realizar la traducción. Ambos tienen la misma función y persiguen lo mismo que es transformar lenguajes fuentes en lenguajes ejecutables.

Los compiladores, primero determinan errores del programa fuente, para luego dejar un programa ejecutable depurado. Se deduce pues que la puesta a punto de un programa es algo más tediosa, ya que hasta que el programa fuente esté perfectamente corregido de errores sintácticos no se podrá traducir. En algunos casos ocurre que partes que nos han producido largas puestas a punto debido a los errores sintácticos que pudieran tener, no serán utilizadas.

Con los intérpretes, cada instrucción se analiza sintácticamente antes de ser ejecutada, es decir, primero el intérprete averigua si hay algún error en la instrucción. Si no lo hay la ejecutará y si lo hay se interrumpirá la ejecución del programa en esa instrucción concreta. Con un compilador esto no se puede realizar, salvo que el programa se ejecute bajo el control de un programa especial de ayuda denominado depurador ("**debugger**"). Por este motivo, los intérpretes resultan más pedagógicos para aprender a programar, ya que el alumno puede detectar y corregir más fácilmente sus errores.

La principal ventaja de los intérpretes frente a los compiladores es que resulta más fácil localizar y corregir errores de los programas, ya que la ejecución de un programa bajo un intérprete puede interrumpirse en cualquier momento para conocer los valores de las distintas variables y la instrucción fuente que acaba de ejecutarse.

El usar intérpretes, sin embargo, hace que la ejecución del programa sea bastante más lenta que con los compiladores, ya que el programa compilado solamente se ejecutará y el programa interpretado además de irse ejecutando, se tendrá que ir analizando a la vez. Además, con los interpretes puede ocurrir que cuando el programa se nos interrumpa por un error sintáctico, se puedan perder datos por no poder relanzar el programa en la instrucción en la que se quedó. Con los compiladores esto no ocurrirá.

10. DEPURADORES

Los depuradores son programas que de forma interactiva permiten localizar errores rápidamente. El depurador pone en marcha el programa ejecutable permitiendo analizar el flujo de ejecución y el estado de los datos cuando van siendo manipulados por el programa.

Existen dos tipos de depuradores según sea la puesta en marcha de los mismos:

- Depuradores integrados en el entorno de compilación.

- Depuradores independientes del entorno del compilador.

Los primeros no necesitan que se haya obtenido el programa ejecutable para poderlo depurar. Una vez obtenido el programa fuente libre de errores de compilación, se ejecutará el depurador desde el mismo entorno, sin necesidad de realizar ninguna acción previa.

Los segundos se ponen en marcha desde el indicador de sistema y hay que indicarles el nombre del programa ejecutable a depurar. Esto implica que el programa fuente ha tenido que ser compilado y enlazado (linkado). Estos depuradores necesitan la presencia del programa fuente en el directorio de trabajo.

El programa depurador, independientemente del tipo, es suministrado por el fabricante, siendo su uso exclusivo para los programas generados por el compilador del mismo fabricante.

Herramientas de un depurador

Dependiendo del fabricante, las operaciones de las que dispone, así como el entorno de presentación, varían de unos a otros. En general las operaciones básicas son:

Rastreo. Permite ver la sentencia (en lenguaje fuente), incluyendo varias opciones:

- Ejecutar una sentencia cada vez, incluso las de procedimientos o funciones por las que pase el flujo de programa.
- Ejecutar una sentencia cada vez del módulo que se está depurando en el momento. Esta opción ejecuta sin parada cualquier llamada a función o procedimiento.
- Ejecutar el programa de forma continua a una velocidad variable y visualizando la sentencia que se está ejecutando.

Punto de parada. Un punto de parada es una pausa que se hace en un lugar determinado dentro del programa. El programa se ejecutará de forma normal hasta llegar al punto de parada, momento en el que el depurador detiene la ejecución del programa y visualiza el código fuente donde se ha detenido.

Ventanas de seguimiento o **monitorización de variables**. En estas ventanas el depurador permite definir cuales de las variables del programa van a estar monitorizadas durante la ejecución del mismo, permitiendo al programador ver la evolución de los valores.

Ejecución controlada. El depurador permite la ejecución continuada hasta el fin del programa desde el punto en el que nos encontremos. También permiten la ejecución hasta la sentencia donde se encuentra el cursor (similar a un punto de parada). Ejecutándose el programa en modo continuado, el depurador permite interrumpir o pausar la ejecución del mismo pulsando la tecla definida para “interrupción del programa” o “pausa”; en este momento sucede lo mismo que se hubiese encontrado un punto de parada.

Reinicialización. El depurador permite reiniciar la ejecución del programa.

Edición de datos y ejecución inmediata de instrucciones. Cuando la ejecución del programa está detenida, el programador puede cambiar los valores de las variables y ejecutar una instrucción no contenida en el programa fuente de modo inmediato.

11. BIBLIOGRAFÍA

Gregorio Fernández
Fundamentos de Informática
Anaya Multimedia, 1995

Alberto Prieto
Introducción a la Informática
Mc Graw-Hill, 2ª edición, 1997

Alfonso Ureña López
Fundamentos de Informática
Ra-ma, 1997