ESCUELA DE PREPARACIÓN DE OPOSITORES
E. P. O.
C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54
30001 MURCIA

INF29 – SAI31

Utilidades para el desarrollo y prueba de programas. Compiladores. Intérpretes. Depuradores.

Esquema.

1 Introducción.

Los lenguajes de programación están específicamente diseñados para programar computadores. Entre sus características destacamos que son independientes de la arquitectura física del computador, y que utilizan notaciones cercanas a las habituales en

el ámbito en que se usan de forma que las operaciones se expresan con sentencias o frases muy parecidas al lenguaje matemático o al lenguaje natural.

Como consecuencia de este alejamiento de la máquina y acercamiento a las personas, los programas escritos en lenguajes de programación no pueden ser directamente interpretados por un computador, siendo necesario realizar previamente su traducción a lenguaje máquina.

En este tema, vamos a estudiar las utilidades para el desarrollo y prueba de programas, estudiando los compiladores, intérpretes y depuradores. Para ello, empezaremos nuestro estudio describiendo el proceso de traducción del código fuente, expresado en un lenguaje de alto nivel, a código máquina.

2 El proceso de traducción.

Para facilitar el uso de los computadores se han desarrollado lenguajes de programación que permiten utilizar una simbología y una terminología próximas a las utilizadas tradicionalmente en la descripción de problemas. Como el computador puede interpretar y ejecutar únicamente código máquina, existen traductores, que traducen programas escritos en lenguajes de programación a lenguaje máquina. Un traductor es un programa que recibe como entrada un texto en un lenguaje de programación concreto y produce, como salida, un texto en lenguaje máquina equivalente. El programa inicial se denomina programa fuente y el programa obtenido, programa objeto.

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 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.

La compilación es un proceso complejo que consume a veces un tiempo muy superior a la propia ejecución del programa. En cualquiera de las fases de análisis, el compilador puede dar mensajes sobre los errores que detecta en el programa fuente, cancelando en ocasiones la compilación, para que el usuario realice las correcciones oportunas en el archivo fuente.

Conceptualmente, un compilador opera en fases, cada una de las cuales transforma al programa fuente de una representación en otra. En la siguiente figura se muestra una descomposición típica de un compilador. En la práctica, se pueden agrupar algunas fases, y las representaciones intermedias entre las fases agrupadas no necesitan ser construidas explícitamente.

2.1 Administración 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 lo 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 de ese registro.

Cuando el analizador léxico detecta un identificador 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.

2.2 Detección e información de errores.

Cada fase 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 porción 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 de 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.

2.3 Análisis lexicográfico.

Consiste en descomponer el programa fuente en sus elementos constituyentes o símbolos (tokens). Los símbolos de un lenguaje son caracteres o secuencias de caracteres que tienen un significado concreto en el lenguaje: cada una de las palabras reservadas, los símbolos de operadores, identificadores de variables, números, etc.

El analizador lexicográfico aísla los símbolos, identifica su tipo y almacena en las tablas de símbolos la información del símbolo que pueda ser necesaria durante el proceso de traducción. Así, por ejemplo, convierte los números o constantes, que en el programa figuran en código de E/S (ASCII, por ejemplo), a su representación interna (entero, coma flotante, precisión simple, etc.), ya que esta información será necesaria a la hora de generar código (no basta con saber que se multiplica por un número, sino que hay que saber su valor).

Podemos considerar que, como resultado del análisis de léxico, se obtiene una representación del programa formada por la descripción de símbolos en las tablas, y una secuencia de símbolos junto con la referencia a la ubicación del símbolo en la tabla. 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. La información almacenada en las tablas de símbolos se completa, y utiliza, en las fases posteriores de la traducción.

Aunque la programación de un analizador de léxico no es compleja, actualmente existen herramientas para generar analizadores de léxico, a partir de una descripción formal del léxico a reconocer.

Por ejemplo, en el análisis léxico los caracteres de la proposición de asignación

```
posición := inicial + velocidad * 60
```

se agruparían en los componentes léxicos siguientes:

1. El identificador `posición`.
2. El símbolo de asignación `:=`.
3. El identificador `inicial`.
4. El signo de suma.
5. El identificador `velocidad`.
6. El signo de multiplicación.
7. El número `60`.

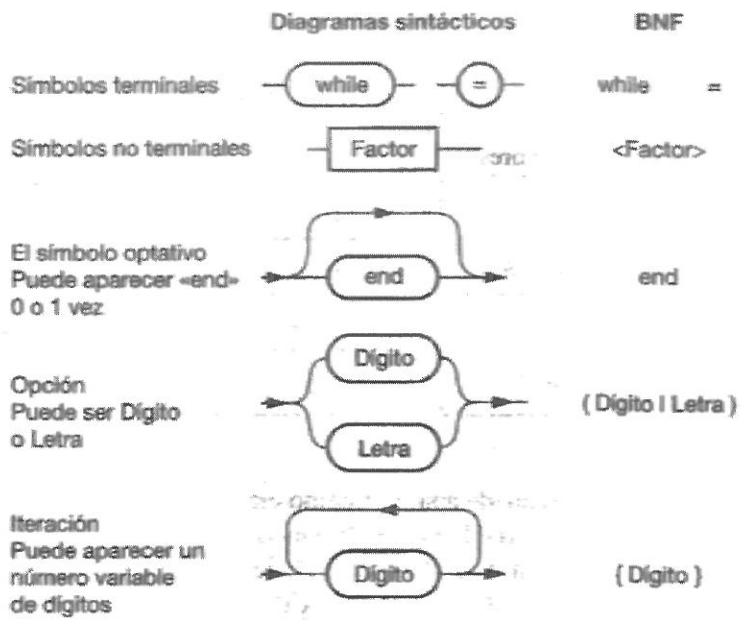
2.4 Análisis sintáctico.

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 (expresiones, sentencias declarativas, asignaciones, etc.) aparecen en un orden correcto. Una construcción puede ser sintácticamente correcta, y carecer de sentido (por ejemplo, asignar un valor entero a una variable tipo cadena de caracteres).

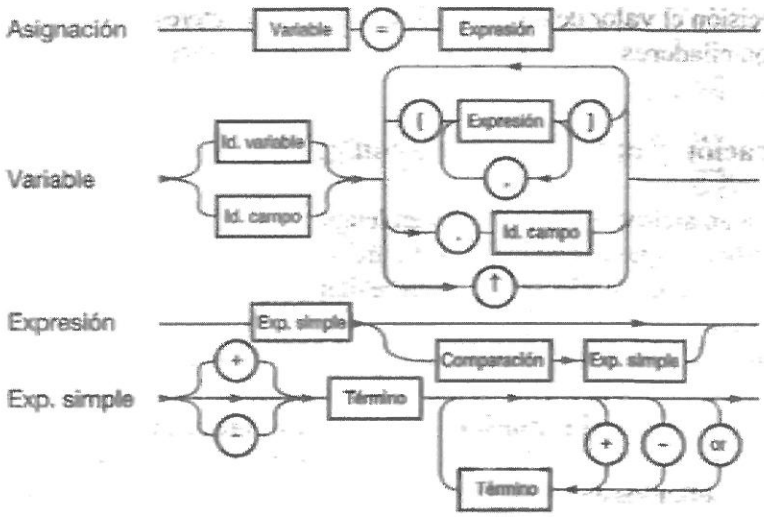
Se han definido varios sistemas para definir la sintaxis de los lenguajes de programación (metalenguajes). Entre ellos cabe destacar la notación BNF (*Backus-Naur-Form*) y los diagramas sintácticos. En ambos casos, se definen construcciones de lenguajes (símbolos no terminales), a partir de los símbolos terminales reconocidos por el analizador de léxico. Un diagrama sintáctico es un grafo dirigido, que describe, al recorrerlo, las distintas posibilidades de creación de una construcción del lenguaje. En BNF, la estructura de una construcción se representa por una expresión del tipo:

```
<Ciclo>::= while <condición> do <bloque>
```

Las construcciones se pueden definir como secuencia de símbolos, repetición de símbolos o elección entre determinados símbolos. La siguiente figura muestra la representación usada, para cada una de estas estructuras, en diagramas sintácticos y en notación BNF.



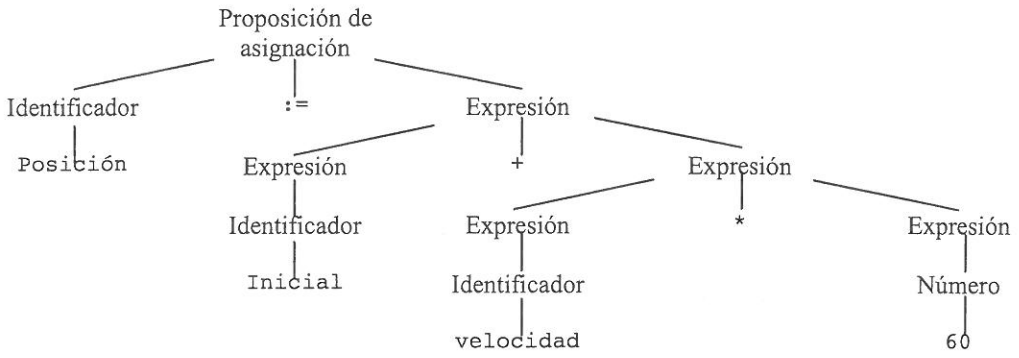
La siguiente figura muestra un ejemplo de la utilización de diagramas sintácticos para describir la sintaxis.



Existe una gran diversidad de métodos para realizar el análisis sintáctico. A un nivel formal, la construcción de un analizador equivale a la construcción de un sistema software que admita como entrada tan sólo las construcciones correctas en el lenguaje. Un método sencillo para construir un analizador es realizar un procedimiento para reconocer cada uno de los símbolos terminales del lenguaje. Cada procedimiento será una transcripción de un diagrama sintáctico, y comprobará si cada nuevo símbolo reconocido puede ser correcto. Esto supone comprobar si el nuevo símbolo está dentro del conjunto de símbolos permitidos en cada momento, y decidir, en base al símbolo aceptado, cuál es ese camino a seguir en el diagrama sintáctico. Este tipo de análisis se conoce como descenso recursivo, dado que el análisis de construcciones recurrentes se realiza con procedimientos recursivos.

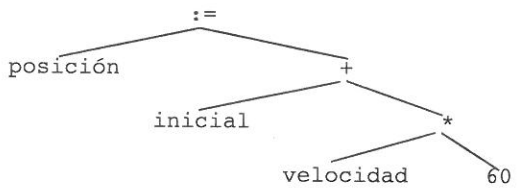
Con este esquema de análisis, el analizador puede generar mensajes de error cuando se encuentre un símbolo no esperado.

El análisis sintáctico implica agrupar los componentes léxicos del programa fuente en frases gramaticales que el compilador utiliza para sintetizar la salida. Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico como el que se ilustra en la siguiente figura.



En la expresión `inicial+velocidad*60`, la frase `velocidad*60` es una unidad lógica, porque las convenciones usuales de las expresiones aritméticas indican que la multiplicación se hace antes que la suma. Puesto que la expresión `inicial+velocidad` va seguida de un `*`, no se agrupa en una sola frase independiente.

El árbol de análisis sintáctico de la figura anterior describe la estructura sintáctica de la entrada. Una representación interna más común de esta estructura sintáctica es la que da el árbol sintáctico de la siguiente figura:



Un árbol sintáctico es una representación compacta del árbol de análisis sintáctico en el que los operadores aparecen como los nodos interiores y los operandos de un operador son los hijos del nodo para ese operador.

2.5 Análisis semántico.

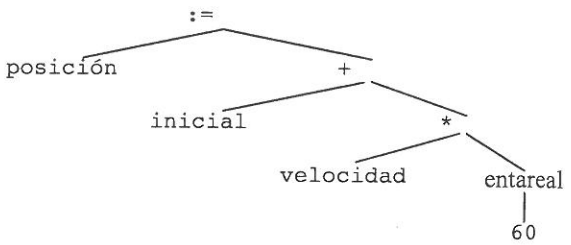
La semántica de un lenguaje de programación es el significado dado a las distintas construcciones sintácticas. El proceso de traducción es, en esencia, la generación de un código en lenguaje máquina con el mismo significado que el código fuente. En los lenguajes de programación, el significado está ligado a la estructura sintáctica de las sentencias. Así, una sentencia de asignación significa transferir el valor de la expresión de la derecha al identificador de la izquierda.

En el proceso de traducción, el significado de las sentencias se obtiene de la identificación sintáctica de las construcciones sintácticas y de la información almacenada en las tablas de símbolos.

Durante la fase de análisis semántico se pueden producir errores, cuando se detectan construcciones "sin un significado correcto". Por ejemplo, asignar a una variable definida como dato numérico en precisión simple el valor de una variable cadena de caracteres, es semánticamente incorrecto para algunos compiladores.

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente. Por ejemplo, las definiciones de muchos lenguajes de programación requieren que el compilador indique un error cada vez que se use un número real como índice de una matriz. Sin embargo, la especificación del lenguaje puede permitir ciertas coerciones a los operandos, por ejemplo, cuando un operador aritmético binario se aplica a un número entero y a un número real. En este caso, el compilador puede necesitar convertir el número entero a real. Así, el árbol de análisis semántico de la figura anterior es:



La estructura jerárquica de un programa normalmente se expresa utilizando reglas recursivas. Por ejemplo, se pueden dar las siguientes reglas como parte de la definición de expresiones:

1. Cualquier identificador es una expresión.
2. Cualquier número es una expresión.
3. Si `expresión`$_1$ y `expresión`$_2$ son expresiones, entonces también lo son

```
expresión1 + expresión2
expresión1 * expresión2
     (expresión1).
```

Las reglas (1) y (2) son reglas básicas (no recursivas), en tanto que la regla (3) define expresiones en función de operadores aplicados a otras expresiones. Así, por la regla (1), `inicial` y `velocidad` son expresiones. Por la regla (2), `60` es una expresión, mientras que por la regla (3), primero podemos inferir que `velocidad*60` es una expresión, y finalmente, que `inicial+velocidad*60` también es una expresión.

De manera similar, muchos lenguajes definen recursivamente las proposiciones mediante reglas como:

1. Si `identificador`$_1$ es un identificador y `expresión`$_2$ es una expresión, entonces también es una proposición:

```
identificador1 := expresión2
```

2. Si `expresión`$_1$ es una expresión y `proposición`$_2$ es una proposición, entonces también son proposiciones:

```
while ( expresión1 ) do proposición2
if ( expresión1 ) then proposición2
```

La división entre análisis léxico y análisis sintáctico es algo arbitraria. Generalmente se elige una división que simplifique la tarea completa del análisis. Un factor para determinar la división es si una construcción del lenguaje fuente es inherentemente recursiva o no. Las construcciones léxicas no requieren recursión, mientras que las construcciones sintácticas suelen requerirla. Las gramáticas independientes del contexto son una formalización de reglas recursivas que se pueden usar para guiar el análisis sintáctico.

Por ejemplo, no se requiere recursión para reconocer los identificadores, que suelen ser cadenas de letras y dígitos que comienzan con una letra. Normalmente, se reconocen los identificadores por el simple examen del flujo de entrada, esperando hasta encontrar un carácter que no sea ni letra ni dígito, y agrupando después todas las letras y dígitos encontrados hasta ese punto en un componente léxico identificador. Los caracteres así agrupados se registran en una tabla llamada tabla de símbolos, y se retiran de la entrada, para que pueda empezar el procesamiento del siguiente elemento léxico.

Por otra parte, esta clase de análisis léxico lineal no es suficientemente poderoso para analizar expresiones o proposiciones. Por ejemplo, no podemos emparejar de manera apropiada los paréntesis de las expresiones, o las palabras `begin` y `end` en proposiciones sin imponer alguna clase de estructura jerárquica o de anidamiento a la entrada.

2.6 Generación y optimización de código.

En esta fase se crea un archivo con un código en lenguaje objeto (normalmente lenguaje máquina) con el mismo significado que el texto fuente. El archivo objeto generado puede ser (dependiendo del compilador) directamente ejecutable, o necesitar otros pasos previos a la ejecución, tales como ensamblado, encadenado y carga. En algunas ocasiones se utiliza un lenguaje intermedio (distinto del código objeto final), con el propósito de facilitar la optimización del código.

En la generación de código intermedio se completan y consultan las tablas generadas en fases anteriores (tablas de símbolos, de constantes, etc.). También se realiza la asignación de memoria a los datos definidos en el programa.

La generación de código puede realizarse añadiendo procedimientos en determinados puntos del proceso de análisis, que generen las instrucciones en lenguaje objeto equivalentes a cada construcción en lenguaje fuente reconocida.

En la fase de optimización se mejora el código intermedio, analizándose el programa objeto globalmente. Un programa puede incluir dentro de un bucle que debe ejecutarse diez mil veces, una sentencia que asigna a una variable un valor constante (B=7.5), no alterándose dicho valor (B) en el bucle. Con ello, innecesariamente se asignaría el valor 7.5 a la variable B diez mil veces. El optimizador sacaría la sentencia B=7.5 fuera (antes) del bucle, ejecutándose así dicha instrucción una sola vez. Hay que hacer notar que el programa inicial es correcto, pero la optimización realizada por el compilador reduce el tiempo de ejecución. Usualmente, las optimizaciones se realizan en varias fases y sobre el código intermedio.

Existen compiladores que permiten al usuario, a su conveniencia, omitir o reducir las fases de optimización, disminuyéndose así el tiempo global de la compilación.

2.6.1 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. Una forma intermedia llamada "código de tres direcciones", 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 anterior puede aparecer en código de tres direcciones como

```
temp1 := entareal(60)
 temp2 := id3 * temp
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 en el programa fuente. 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 operandos, por ejemplo, la primera y la última instrucción.

En general, estas representaciones deben hacer algo más que calcular expresiones; también deben manejar construcciones de flujo de control y llamadas a procedimientos.

2.6.2 Optimización de código.

La fase de optimización de código trata de mejorar el código intermedio, de modo que resulte un código de máquina más rápido de ejecutar. Algunas optimizaciones son triviales. Por ejemplo, un algoritmo natural genera el código intermedio anterior utilizando una instrucción para cada operador de la representación de árbol después del análisis semántico, aunque hay una forma mejor de realizar los mismos cálculos usando las dos instrucciones

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Este sencillo algoritmo no tiene nada de malo, puesto que el problema se puede solucionar en la fase de optimización 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 `entareal` se puede eliminar. Además, `temp3` se usa sólo una vez, para transmitir su valor a `id1`. Entonces resulta seguro sustituir `id1` por `temp3`.

Hay mucha variación en la cantidad de optimización de código que ejecutan los distintos compiladores. En los que hacen mucha optimización, llamados "compiladores optimizadores", una parte significativa del tiempo del compilador se ocupa en esta fase. Sin embargo, hay optimizaciones sencillas que mejoran sensiblemente el tiempo de ejecución del programa objeto sin retardar demasiado la compilación.

2.6.3 Generación de código.

La fase final de un compilador es la generación de código objeto, que por lo general consiste en código de máquina relocalizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asignación de variables a registros.

Por ejemplo, utilizando los registros 1 y 2, la traducción del código anterior podría convertirse en:

```
MOVF id3,R2
MULF #60.0,R2
MOVF id2,R1
ADDF R2,R1
MOVF R1,id1
```

El primero y segundo operandos de cada instrucción especifican una fuente y un destino, respectivamente. La `F` de cada instrucción indica que las instrucciones trabajan con números de punto flotante. Este código traslada el contenido de la dirección `id3` al registro 2, después lo multiplica por la constante real `60.0`. El signo # significa que `60.0` se trata como una constante. La tercera instrucción pasa `id2` al registro 1. La

cuarta instrucción le suma el valor previamente calculado en el registro 2. Por último, el valor del registro 1 se pasa a la dirección de `id1`, de modo que el código aplica la asignación.

2.7 Las fases de análisis.

Conforme avanza la traducción, la representación interna del programa fuente que tiene el compilador se modifica. Para ilustrar esas representaciones, consideremos globalmente el ejemplo anterior:

```
posición := inicial + velocidad * 60
```

La siguiente figura muestra la representación de esta proposición después de cada fase.

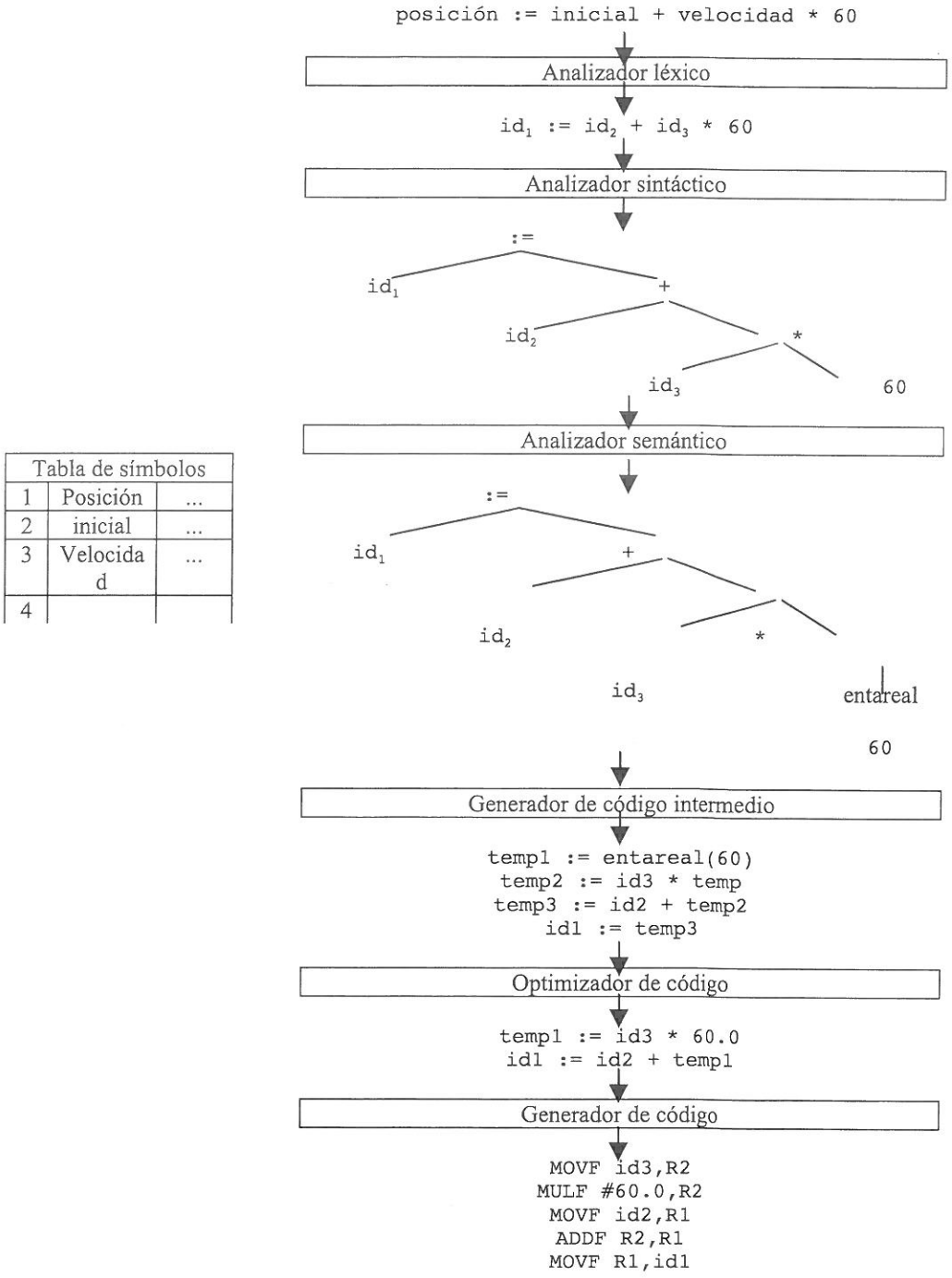


Tabla de símbolos		
1	Posición	...
2	inicial	...
3	Velocidad	...
4		

La fase de análisis léxico lee los caracteres en el 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`, etcétera), un carácter de puntuación, o un operador de varios caracteres, como `:=`. La secuencia de caracteres que forma 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`.

En esta sección, se usarán 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.

El análisis sintáctico impone una estructura jerárquica a la cadena de componentes léxicos, que se representará por medio de árboles sintácticos.

2.8 El agrupamiento de las fases.

En una implementación, a menudo se agrupan las actividades en dos o más fases.

2.8.1 Etapa inicial y etapa final.

Con frecuencia, las fases se agrupan en una etapa inicial y una etapa final. La etapa inicial comprende aquellas fases, o partes de fases, que dependen principalmente del lenguaje fuente y que son en gran parte independientes de la máquina objeto. Ahí normalmente se incluyen los análisis léxico y sintáctico, la creación de la tabla de símbolos, el análisis semántico y la generación de código intermedio. La etapa inicial también puede hacer cierta optimización de código. La etapa inicial incluye, además, el manejo de errores correspondiente a cada una de esas fases.

La etapa final incluye aquellas partes del compilador que dependen de la máquina objeto y, en general, esas partes no dependen del lenguaje fuente, sino sólo del lenguaje intermedio. En la etapa final, se encuentran aspectos de la fase de optimización de código, además de la generación de código, junto con el manejo de errores necesario y las operaciones con la tabla de símbolos.

Se ha convertido en rutina el tomar la etapa inicial de un compilador y rehacer su etapa final asociada para producir un compilador para el mismo lenguaje fuente en una máquina distinta. Si la etapa final se diseña con cuidado, incluso puede no ser necesario rediseñarla demasiado. También resulta tentador compilar varios lenguajes distintos en el mismo lenguaje intermedio y usar una etapa final común para las distintas etapas iniciales, obteniéndose así varios compiladores para una máquina. Sin embargo, dadas las sutiles diferencias en los puntos de vista de los distintos lenguajes, sólo se ha obtenido un éxito limitado en ese aspecto.

2.8.2 Pasadas.

Normalmente se aplican varias fases de la compilación en una sola pasada, que consiste en la lectura de un archivo de entrada y en la escritura de un archivo de salida. En la práctica, hay muchas formas de agrupar en pasadas las fases de un compilador, así

que es preferible organizar el análisis de la compilación por las fases, en lugar de por las pasadas.

Corno ya hemos señalado, es común agrupar varias fases en una pasada, y entrelazar la actividad de estas fases durante la pasada. Por ejemplo, el análisis léxico, el análisis sintáctico, el análisis semántico y la generación de código intermedio pueden agruparse en una pasada. En ese caso, la cadena de componentes léxicos después del análisis léxico puede traducirse directamente a código intermedio. Con más detalle, el analizador sintáctico puede considerarse como el "encargado" del control. Éste intenta descubrir la estructura gramatical de los componentes léxicos observados; obtiene los componentes léxicos cuando los necesita, llamando al analizador léxico para que le proporcione el siguiente componente léxico. A medida que se descubre la estructura gramatical, el analizador sintáctico llama al generador de código intermedio para que haga el análisis semántico y genere una parte del código.

2.8.3 Reducción del número de pasadas.

Es deseable tener relativamente pocas pasadas, dado que la lectura y escritura de archivos intermedios lleva tiempo. Además, si se agrupan varias fases dentro de una pasada, puede ser necesario tener que mantener el programa completo en memoria, porque una fase puede necesitar información en un orden distinto al que produce una fase previa. La forma interna del programa puede ser considerablemente mayor que el programa fuente o el programa objeto, de modo que este espacio no es un tema trivial.

Para algunas fases, el agrupamiento en una pasada presenta pocos problemas. Por ejemplo, como mencionamos antes, la interfaz entre los analizadores léxico y sintáctico a menudo puede limitarse a un solo componente léxico. Por otra parte, muchas veces resulta muy difícil generar código hasta que se haya generado por completo la representación intermedia. No se puede generar el código objeto para una construcción si no se conocen los tipos de las variables implicadas en esa construcción. De manera similar, la mayoría de los lenguajes admiten construcciones `goto` que saltan hacia adelante en el código. No se puede determinar la dirección objeto de dichos saltos hasta haber visto el código fuente implicado y haber generado código objeto para él.

En algunos casos, es posible dejar un segmento en blanco para la información que falta, y llenar la ranura cuando la información esté disponible. En particular, la generación de código intermedio y de código objeto a menudo se pueden fusionar en una sola pasada utilizando una técnica llamada "relleno de retroceso" (*backpatching*).

3 Utilidades para el desarrollo y prueba de programas.

3.1 Compiladores e intérpretes.

Un compilador traduce 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 archivo, y el programa objeto puede almacenarse como archivo 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.

Un intérprete hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador. 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 almacenable en memoria masiva para ulteriores ejecuciones. Por tanto, la ejecución del programa está 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 almacenando 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, de forma que:

1. Si una sentencia forma parte de un bucle, se analiza tantas veces como tenga que ejecutarse el bucle.
2. Las optimizaciones sólo se realizan dentro del contexto de cada sentencia, y no contemplándose el programa o sus estructuras en conjunto.
3. Cada vez que utilicemos un programa tenemos que volver a analizarlo, ya que en la traducción no se genera un archivo objeto que poder guardar en memoria masiva (y utilizarlo en cada ejecución). Con un compilador, aunque la traducción sea más lenta, ésta sólo debe realizarse una vez (ya depurado el programa) y cuando deseamos ejecutar un programa, ejecutamos el archivo objeto, que se tradujo previamente.

Los intérpretes, a pesar de los inconvenientes anteriores, son preferibles a los compiladores cuando el número de veces que se va a ejecutar el programa es muy bajo y no hay problemas de velocidad; además, con ellos puede ser más fácil desarrollar programas. Esto es así porque, normalmente, 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. Cuando esto se hace, se tiene una gran ayuda para la localización de errores en el programa. Con un programa compilador esto no se puede realizar, salvo que el programa se ejecute bajo el control de un programa especial de ayuda denominado depurador (*debugger*). Además, con un intérprete, cuando se localiza un error sólo debe modificarse este error y volver a ejecutar a partir de la instrucción en cuestión. Con un compilador, si se localiza un error durante la ejecución, el programador debe corregir las instrucciones erróneas sobre el archivo fuente (no sobre el objeto), y volver a compilarlo en su totalidad.

En lugar de producir un programa objeto como resultado de una traducción, un intérprete realiza las operaciones que implica el programa fuente. Para una proposición de asignación, por ejemplo, un intérprete podría construir un árbol, y después efectuar las operaciones de los nodos conforme "recorre" el árbol. Muchas veces los intérpretes se usan para ejecutar lenguajes de órdenes, pues cada operador que se ejecuta en un lenguaje de órdenes suele ser una invocación de una rutina compleja, como un editor o un compilador. Del mismo modo, algunos lenguajes de "muy alto nivel", como APL, normalmente son interpretados, porque hay muchas cosas sobre los datos, como el tamaño y la forma de las matrices, que no se pueden deducir en el momento de la compilación.

Existen traductores que en cierta medida disponen de las ventajas tanto de los intérpretes, como de los compiladores. Estos traductores se denominan compiladores interactivos o incrementales. Un compilador interactivo es un sistema que permite la

edición, compilación, ejecución y depuración de programas escritos en un determinado lenguaje de alto nivel. Pueden presentar esencialmente las siguientes ventajas:

- Es muy cómoda la creación y depuración del programa, puesto que la iteración edición-compilación-ejecución no implica ejecutar distintos programas desde el sistema operativo.
- Permite que el compilador, durante la edición, realice parte de las comprobaciones, y en algunos casos la traducción a código intermedio, lo que permite detectar con antelación la mayor parte de los tipos de error.
- Cuando se realiza una modificación puede no ser necesario retraducir todo el programa, si no tan solo la parte implicada en el cambio (de ahí el nombre de incrementales).

3.2 Traductores cruzados. Emuladores.

Se denominan traductores cruzados aquellos traductores que efectúan la traducción de programas fuente a programas objeto en un computador distinto (computador B) a aquel en el que se ejecutará el programa objeto (computador A). El computador B se denomina computador anfitrión (host) y el computador A, computador huésped (*guest*). Por lo general el computador anfitrión es más potente que el huésped.

También en un computador puede simularse el comportamiento de otro. Estos programas de simulación se suelen denominar emuladores.

3.3 Herramientas de análisis.

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 del programa fuente. La parte de la síntesis construye el programa objeto deseado a partir de la representación intermedia. De las dos partes, la síntesis es la que requiere las técnicas más especializadas.

Durante el análisis, se determinan las operaciones que implica el programa fuente y se registran 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.

Muchas herramientas de software que manipulan programas fuente realizan primero algún tipo de análisis. Algunos ejemplos de tales herramientas son: editores de estructuras, impresoras estéticas, y verificadores estáticos,

3.3.1 Editores de estructuras.

Un editor de estructuras toma como entrada una secuencia de órdenes 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 (por ejemplo, cuando el usuario escribe `while`, el editor proporciona el correspondiente `do` y le recuerda al usuario que entre las dos palabras debe ir un condicional) 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 de análisis de un compilador.

3.3.2 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.

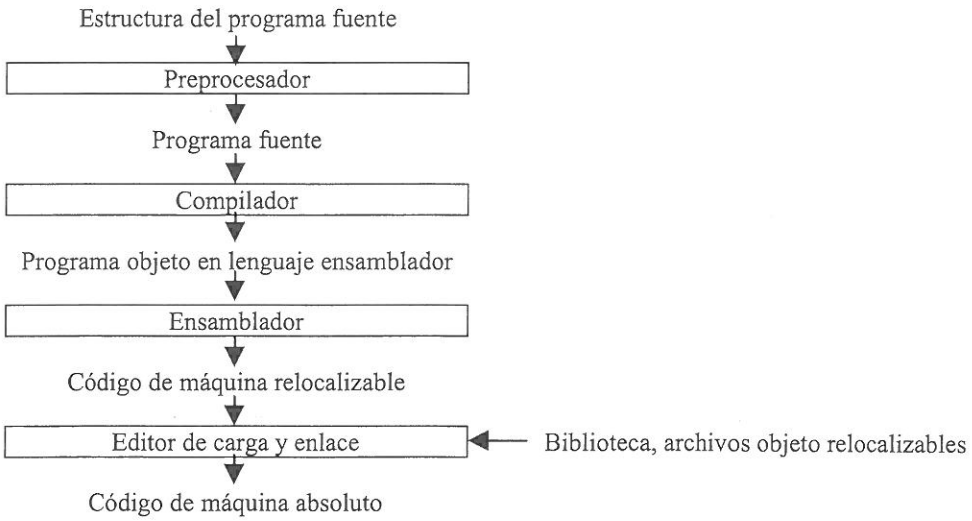
3.3.3 Verificadores estáticos.

Un verificador estático lee un programa, lo analiza e intenta descubrir errores potenciales sin ejecutar el programa. La parte del análisis a menudo es similar a la que se encuentra en los compiladores de optimización. 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 puntero, empleando las técnicas de verificación de tipos.

3.4 Programas de sistemas relacionados con un compilador.

Además de un compilador, se pueden necesitar otros programas para crear un programa objeto ejecutable. Un programa fuente se puede dividir en módulos almacenados en archivos distintos. La tarea de reunir el programa fuente a menudo se confía a un programa distinto, llamado preprocesador. El preprocesador también puede expandir abreviaturas, llamadas macros, a proposiciones del lenguaje fuente.

La siguiente figura muestra una "compilación" típica. El programa objeto creado por el compilador puede requerir procesamiento adicional antes de poderlo ejecutar. El compilador de la figura crea código en lenguaje ensamblador el cual es traducido por un ensamblador a código de máquina y después se enlaza a algunas rutinas de biblioteca para producir el código que realmente se ejecute en la máquina.



La entrada para un compilador puede producirse por uno o varios preprocesadores, y puede necesitarse otro procesamiento de la salida que produce el compilador antes de obtener un código de máquina ejecutable. En esta sección se analiza el contexto en el que suele funcionar un compilador.

3.4.1 Preprocesadores.

Los preprocesadores producen la entrada para un compilador, y pueden realizar las funciones siguientes:

1. Procesamiento de macros. Un preprocesador puede permitir a un usuario definir macros, que son abreviaturas de construcciones más grandes.
2. Inclusión de archivos. Un preprocesador puede insertar archivos de encabezamiento en el texto del programa. Por ejemplo, el preprocesador de C hace que el contenido del archivo <global. h> reemplace a la proposición `#include <global.h>` cuando procesa un archivo que contenga a esa proposición.
3. Preprocesadores "racionales". Estos preprocesadores enriquecen los lenguajes antiguos con recursos más modernos de flujo de control y de estructuras de datos. Por ejemplo, un preprocesador de este tipo podría proporcionar al usuario macros incorporadas para construcciones, como proposiciones `while` o `if`, en un lenguaje de programación que no las tenga.
4. Extensiones a lenguajes. Estos procesadores tratan de crear posibilidades al lenguaje que equivalen a macros incorporadas.

Los procesadores de macros tratan dos clases de proposiciones: definición de macros y uso de macros. Las definiciones normalmente se indican con algún carácter exclusivo o palabra clave, como `define` o `macro`. Constan de un nombre para la macro que se está definiendo y de un cuerpo, que constituye su definición. A menudo, los procesadores de macros admiten parámetros formales en su definición, esto es, símbolos que se reemplazarán por valores (en este contexto, un "valor" es una cadena de caracteres). El uso de una macro consiste en dar nombre a la macro y proporcionar parámetros reales, es decir, valores para sus parámetros formales. El procesador de macros sustituye los parámetros reales por los parámetros formales del cuerpo de la macro; después, el cuerpo transformado reemplaza el uso de la propia macro.

3.4.2 Ensambladores.

Algunos compiladores producen código ensamblador. Otros compiladores realizan el trabajo del ensamblador, produciendo código de máquina relocalizable que se puede pasar directamente al editor de carga y enlace.

El código ensamblador es una versión mnemotécnica del código de máquina, donde se usan nombres en lugar de códigos binarios para operaciones, y también se usan nombres para las direcciones de memoria. Una secuencia típica de instrucciones en ensamblador puede ser:

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

Este código pasa el contenido de la dirección `a` al registro 1; después le suma la constante 2, tratando al contenido del registro 1 como un número de punto fijo, y por último almacena el resultado en la posición de memoria que representa `b`. De ese modo, calcula `b := a + 2`.

Es común que los lenguajes ensambladores tengan recursos para manejar macros que son similares a las consideradas antes para los preprocesadores de macros.

3.4.3 Ensamblado de dos pasadas.

La forma más simple de un ensamblador hace dos pasadas sobre la entrada, en donde una pasada consiste en leer una vez un archivo de entrada. En la primera pasada, se encuentran todos los identificadores que denoten posiciones de memoria y se almacenan en una tabla de símbolos (distinta de la del compilador). Cuando se encuentran por primera vez los identificadores, se les asignan posiciones de memoria.

En la segunda pasada, el ensamblador examina el archivo de entrada de nuevo. Esta vez traduce cada código de operación a la secuencia de bits que representa esa operación en lenguaje de máquina, y traduce cada identificador que representa una posición de memoria a la dirección dada por ese identificador en la tabla de símbolos.

El resultado de la segunda pasada normalmente es código de máquina relocalizable, lo cual significa que puede cargarse empezando en cualquier posición L de la memoria; es decir, si se suma L a todas las direcciones del código, entonces todas las referencias serán correctas. Por tanto, la salida del ensamblador debe distinguir aquellas partes de instrucciones que se refieren a direcciones que se pueden reubicar.

3.4.4 Cargadores y editores de enlace.

Por lo general, un programa llamado cargador realiza las dos funciones de carga y edición de enlaces. El proceso de carga consiste en tomar el código de máquina relocalizable, modificar las direcciones reubicables, y ubicar las instrucciones y los datos modificados en las posiciones apropiadas de la memoria.

El editor de enlace permite formar un sólo programa a partir de varios archivos de código de máquina relocalizable. Estos archivos pueden haber sido el resultado de varias compilaciones distintas, y uno o varios de ellos pueden ser archivos de biblioteca de rutinas proporcionadas por el sistema y disponibles para cualquier programa que las necesite.

Si los archivos se van a usar juntos de manera útil, puede haber algunas referencias externas, en las que el código de un archivo hace referencia a una posición de otro archivo. Esta referencia puede ser a una posición de datos definida en un archivo y utilizada en otro, o puede ser el punto de entrada de un procedimiento que aparece en el código de un archivo y se llama desde otro. El archivo con el código de máquina relocalizable debe conservar la información de la tabla de símbolos para cada posición de datos o etiqueta de instrucción a la que se hace referencia externamente. Si no se sabe por anticipado a qué se va a hacer referencia, es preciso incluir completa la tabla de símbolos del ensamblador como parte del código de máquina relocalizable.

3.5 Depuradores.

Mientras se realiza el proceso de desarrollo de un programa, resulta muy interesante conocer cómo se ejecuta y qué ocurre en cada momento en nuestro programa. Para ello, algunos sistemas poseen unos programas depuradores cuya misión es permitir la ejecución paso a paso o por tramos del programa manteniendo el entorno que se va produciendo (valores de variables). El programador en cada parada de la ejecución de su programa puede comprobar e incluso modificar valores de las variables, con lo cual resulta muy interesante este tipo de ejecución para la detección de errores y comprobación del buen o mal funcionamiento del programa. Los puntos de parada en la ejecución son determinados por el propio programador mediante sentencias destinadas a tal fin.

4 Conclusiones.

Para facilitar el uso de los computadores se han desarrollado lenguajes de programación que permiten utilizar una simbología y una terminología próximas a las utilizadas tradicionalmente en la descripción de problemas. Como el computador puede interpretar y ejecutar únicamente código máquina, existen traductores, que traducen programas escritos en lenguajes de programación a lenguaje máquina. Un traductor es un programa que recibe como entrada un texto en un lenguaje de programación concreto y produce, como salida, un texto en lenguaje máquina equivalente.

La traducción por un compilador (la compilación) consta de dos etapas fundamentales, la etapa de análisis del programa 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.

Un compilador traduce un programa fuente, escrito en un lenguaje de alto nivel, a un programa objeto, escrito en lenguaje ensamblador o máquina. El programa objeto puede almacenarse como archivo en memoria masiva para ser procesado posteriormente, sin necesidad de volver a realizar la traducción.

Un intérprete hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador. 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 almacenable en memoria masiva para ulteriores ejecuciones.

Un traductor cruzado es el traductor que efectúa la traducción en un computador distinto a aquel en el que se ejecutará el programa objeto.

Existen herramientas software que manipulan programas fuente realizando algún tipo de análisis. Algunos ejemplos de tales herramientas son: editores de estructuras, impresoras estéticas, y verificadores estáticos.

Además de un compilador, se pueden necesitar otros programas para crear un programa objeto ejecutable. Entre éstos, destacan: el preprocesador, ensamblador, y los cargadores y editores de enlace.

Los sistemas, normalmente, poseen programas depuradores cuya misión es permitir la ejecución paso a paso o por tramos del programa manteniendo el entorno que se va produciendo (valores de variables). El programador puede comprobar e incluso modificar valores de las variables, pudiendo detectar errores y comprobar el funcionamiento del programa.