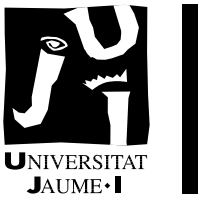


4º Ingeniería Informática

II26 Procesadores de lenguaje

Estructura de los compiladores e intérpretes

Esquema del tema

1. Introducción
2. Etapas del proceso de traducción
3. La interpretación
4. La arquitectura real de compiladores e intérpretes
5. Resumen del tema

1. Introducción

Tanto los compiladores como los intérpretes son programas de gran complejidad. Afortunadamente, se sabe suficiente acerca de cómo estructurarlos y hay suficientes herramientas formales para que la complejidad se reduzca a niveles razonables. En este tema veremos en qué fases se divide un compilador o un intérprete. Veremos también qué tienen en común y cómo difieren entre sí compiladores e intérpretes.

2. Etapas del proceso de traducción

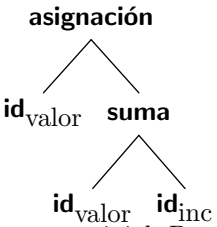
Podemos modelar el proceso de traducción entre dos lenguajes como el resultado de dos etapas. En la primera etapa se analiza la entrada para averiguar qué es lo que se intenta comunicar. Esto es lo que se conoce como *análisis*. El fruto de esta etapa es una representación de la entrada que permite que la siguiente etapa se desarrolle con facilidad. La segunda etapa, la *síntesis*, toma la representación obtenida en el análisis y la transforma en su equivalente en el lenguaje destino.

En el caso de la interpretación, se utiliza la representación intermedia para obtener los resultados deseados.

2.1. Análisis

El objetivo de esta etapa es obtener una representación de la entrada que nos permita realizar la síntesis o la interpretación con comodidad. La representación que nosotros utilizaremos es la que se llama *árbol de sintaxis abstracta*. Un ejemplo sería la traducción siguiente:

```
valor= valor+inc; /* Actualizamos */
```



El paso de la entrada al árbol de sintaxis abstracta no es trivial. Para facilitarlo, se divide la tarea en varias partes. Supón que tuvieras que describir un lenguaje de programación. Una manera de hacerlo sería comenzando por describir cuáles son las unidades elementales tales como identificadores, palabras reservadas, operadores, etc. que se encuentran en la entrada. Después podrías describir cómo se pueden combinar esas unidades en estructuras mayores tales como expresiones, asignaciones, bucles y demás. Finalmente, especificarías una serie de normas que deben cumplirse para que el programa, además de estar "bien escrito", tenga significado. Estas normas se refieren

a aspectos tales como que las variables deben declararse o las reglas que se siguen para decidir los tipos de las expresiones.

Las tres fases que hemos mencionado tienen su reflejo en las tres fases en que se divide el análisis:

Análisis léxico: se encarga de la división de la entrada en componentes léxicos.

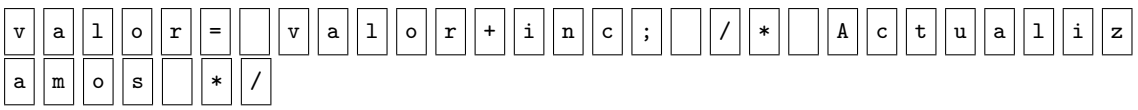
Análisis sintáctico: se encarga de encontrar las estructuras presentes en la entrada.

Análisis semántico: se encarga de comprobar que se cumplen las restricciones semánticas del lenguaje.

2.1.1. Análisis léxico

En esta fase se analiza la entrada carácter a carácter y se divide en una serie de unidades elementales: los *componentes léxicos*. Cada uno de estos componentes se clasifica en una categoría y puede recibir uno o más atributos con información relevante para otras fases (por ejemplo un entero tendría una etiqueta indicando su valor). El criterio que se emplea para clasificar cada componente es su pertenencia o no a un lenguaje (generalmente regular). Esta fase además se encarga de filtrar elementos tales como los blancos y los comentarios.

En nuestro ejemplo, tendríamos como categorías los identificadores, la suma, la asignación y el punto y coma. Podemos suponer que los identificadores son secuencias de letras y dígitos que comienzan por una letra. Además, hay otros componentes que "se filtran" o, más formalmente, son omitidos: los blancos y los comentarios. Teniendo en cuenta esto, nuestro analizador léxico ve:



Y lo que pasa al sintáctico es:

Donde hemos asumido que **id** es el componente léxico que representa los identificadores; **asig** representa la asignación; **suma**, las sumas y **pyc**, el punto y coma. Como puedes ver, han desaparecido tanto los blancos como los comentarios.

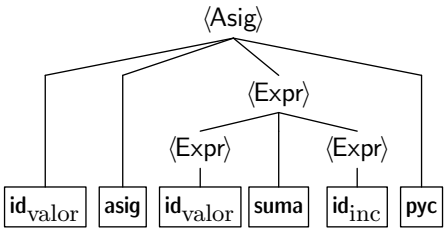
2.1.2. Análisis sintáctico

Partiendo de lo que ha recibido del analizador léxico, la tarea del analizador sintáctico consiste en ir descubriendo las estructuras presentes en el código de acuerdo con una gramática incontextual. A partir de las estructuras que ha encontrado, el analizador sintáctico construye un *árbol sintáctico* (que no hay que confundir con el árbol de sintaxis abstracta comentado antes).

Para especificar las construcciones que se permiten, se suelen emplear gramáticas incontextuales, que veremos luego. En nuestro caso podemos pensar que las reglas que se siguen son que una asignación se compone de un identificador, seguido de un símbolo de asignación, seguido de una expresión y de un punto y coma. Esto se escribe en la gramática en forma de regla: ⟨Asig⟩ → **id asig** ⟨Expr⟩ **pyc**. Análogamente, podemos decir que una expresión es bien un identificador, bien la suma de dos expresiones. En reglas:

$$\begin{aligned} \langle \text{Expr} \rangle &\rightarrow \textbf{id} \\ \langle \text{Expr} \rangle &\rightarrow \langle \text{Expr} \rangle \ \textbf{suma} \ \langle \text{Expr} \rangle \end{aligned}$$

El árbol sintáctico nos permite expresar cómo se puede "fabricar" (formalmente, derivar) la entrada a partir de las reglas. En nuestro caso, el árbol tiene un aspecto similar a:



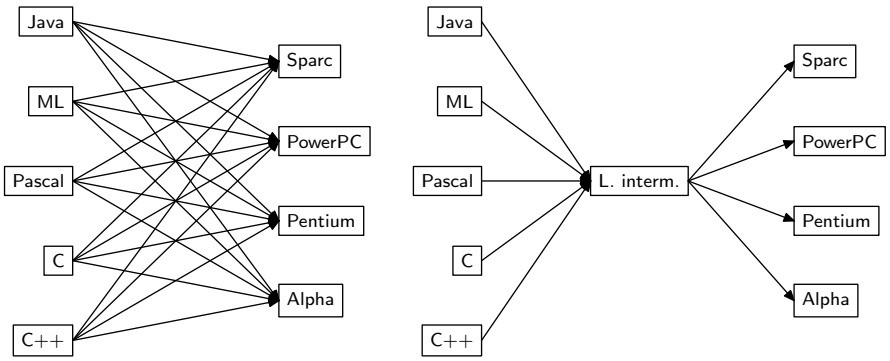
Es interesante darse cuenta de que, tanto en las reglas como en la construcción del árbol, se hace caso omiso de los posibles atributos de los componentes léxicos; únicamente se tiene en cuenta su categoría.

2.1.3. Análisis semántico

La última fase del análisis, el análisis semántico, toma como entrada el árbol sintáctico y comprueba si, además de las restricciones sintácticas, se cumplen otras restricciones impuestas por el lenguaje y que no pueden ser comprobadas mediante una gramática incontextual. Algunos ejemplos de estas restricciones son la necesidad de declarar las variables antes de usarlas, las reglas de tipos o la coincidencia entre los parámetros de las funciones en las definiciones y las llamadas. Como salida de esta fase, se obtiene una representación semántica, por ejemplo el árbol de sintaxis abstracta comentado antes. Además, se ha comprobado que tanto `valor` como `inc` están declaradas y con tipos compatibles.

2.2. Síntesis

Una vez analizado el programa de entrada, es necesario generar código, a ser posible eficiente, para la máquina objetivo. Supongamos que tenemos L lenguajes fuente y queremos escribir compiladores para M máquinas distintas. La aproximación inmediata, escribir un compilador para cada par lenguaje-máquina, supone escribir $L \times M$ compiladores. Sin embargo, si los lenguajes son razonablemente parecidos (como Pascal y C), existe una aproximación mejor: escribir L traductores desde los lenguajes fuente a un lenguaje intermedio y después escribir M traductores de este lenguaje intermedio a los lenguajes máquina correspondientes:



Esta aproximación tiene diversas ventajas. La más obvia es la reducción del número de traductores que se necesitarán. Además, si se quiere añadir un nuevo lenguaje a nuestra colección, no es necesario crear M compiladores para él, basta con un traductor al lenguaje intermedio. Esto permite que se desarrollen nuevos lenguajes con comodidad. Más importante, si aparece una nueva arquitectura, basta con desarrollar un traductor del lenguaje intermedio a esta nueva máquina.

Otro aspecto de gran importancia es que la representación intermedia suele elegirse de modo que no necesite el gran nivel de detalle que exige el código máquina y permita abstraer problemas como el número limitado de registros disponibles o la gran variedad de instrucciones de máquina donde elegir. Esto simplifica notablemente la generación de código desde el AST.

Desgraciadamente, esta aproximación no es gratis ni universal. Encontrar un lenguaje intermedio que sea adecuado para todas las arquitecturas y lenguajes no es tarea sencilla y probablemente no sea posible. En la práctica se emplea para lenguajes similares y máquinas destino que compartan una serie de características comunes. Por otro lado, determinadas características del par (lenguaje fuente, máquina destino) pueden no aprovecharse. Por ejemplo, si sabemos que la máquina destino tiene instrucciones especializadas para saltos mediante una tabla, el código generado para las estructuras `switch-case` debería reflejarlo. Esto no es posible si el lenguaje intermedio no recoge este tipo de construcciones.

Pese a todo, las ventajas superan a los inconvenientes y la generación de código se divide habitualmente en dos etapas:

- Generación de código intermedio.
- Generación de código objeto: se traduce el código intermedio a código de máquina.

2.2.1. Generación de código intermedio

En esta etapa se traduce la entrada a una representación independiente de la máquina pero fácilmente traducible a lenguaje ensamblador. Esta representación puede tomar diversas formas que pueden entenderse como visiones idealizadas del lenguaje ensamblador de una máquina virtual. Algunas de las representaciones más comunes son:

- Árboles de representación intermedia (distintos de los árboles de sintaxis abstracta),
- código de tres direcciones,
- código de dos direcciones,
- código de pila,
- representaciones en forma de grafo, mixtas, etc...

En nuestro caso, usando la máquina virtual de las prácticas, la sentencia `valor= valor+inc;` puede traducirse por algo similar a:

```
lw $r0, -2($fp)   # Carga valor en $r0
lw $r1, -1($fp)   # Carga inc en $r1
add $r0, $r0, $r1 # Hace la suma
sw $r0, -2($fp)   # Guarda el resultado en valor
```

2.2.2. Generación de código objeto

Una vez obtenido el código intermedio, es necesario generar el código objeto. Lo habitual es que no se genere el código objeto directamente sino que se genere código en ensamblador y después se utilice un ensamblador. De cualquier forma, esta fase es totalmente dependiente de la arquitectura concreta para la que se esté desarrollando el compilador. En particular, hay que enfrentarse a problemas como:

- Selección de instrucciones teniendo en cuenta su eficiencia.
- Elección de los modos de direccionamiento adecuados.
- Utilización eficiente de los registros.
- Empleo eficiente de la caché.
- Otros...

Las instrucciones que genera `gcc` para nuestro ejemplo son:

```
movl    -8(%ebp), %edx
leal    -4(%ebp), %eax
addl    %edx, (%eax)
```

Como ves, se tiene en cuenta el tamaño de las variables (por eso se emplea -4 y -8 en lugar de -2 y -1), se utilizan registros de la máquina concreta y se emplean instrucciones especiales como `leal` para aprovechar mejor el procesador.

2.2.3. Optimización

Tanto a la hora de generar código intermedio como código objeto es habitual encontrarse con que el resultado de la traducción es muy ineficiente. Esto es debido a que la traducción se realiza de manera local, lo cual provoca la aparición de código redundante. Por ejemplo, la sentencia `a[i]=a[i]+1` genera el siguiente código intermedio:

```
addi $r0, $zero, 0      # Dirección de a
lw $r1, 3($zero)        # Acceso a i
add $r0, $r0, $r1       # Sumamos i a la dirección de a
lw $r0, 0($r0)          # Valor de a[i]
addi $r1, $zero, 1      # Valor entero (1)
add $r0, $r0, $r1       # Hacemos la suma
addi $r1, $zero, 0      # Dirección de a
lw $r2, 3($zero)        # Acceso a i
add $r1, $r1, $r2       # Sumamos i a la dirección de a
sw $r0, 0($r1)          # Guardamos el resultado
```

Sin embargo, es fácil darse cuenta de que no hace falta calcular dos veces la dirección de `a[i]`, con lo que se pueden ahorrar, al menos, tres instrucciones.

Ejercicio 1

Escribe el código de menor longitud que se te ocurra para el ejemplo anterior. No puedes utilizar instrucciones distintas de las mostradas.

En otras ocasiones, es posible utilizar instrucciones especializadas para mejorar la velocidad. Por ejemplo, si la instrucción anterior a `valor= valor+inc;` es `inc=1;`, `gcc -O` genera

```
incl    %ebx
```

Por esto, es habitual incluir módulos encargados tanto de la optimización del código intermedio como del código objeto.

2.3. Otros módulos del compilador

Aunque en principio sería posible escribir el compilador como la concatenación de las distintas fases que se han descrito, existen dos módulos que no forman parte de esta secuencia pero tienen un papel fundamental para el proceso de compilación: la tabla de símbolos y el módulo de gestión de errores.

2.3.1. La tabla de símbolos

A lo largo del proceso de análisis se va generando gran cantidad de información que se puede considerar ligada a los objetos que se van descubriendo en el programa: variables, constantes, funciones, etc. El acceso a esta información se realiza mediante los nombres de estos objetos. Esto hace necesario tener alguna manera de, a partir de un identificador, encontrar sus propiedades. La estructura de datos que guarda esta información se denomina *tabla de símbolos* y puede interactuar con prácticamente todas las fases de la compilación.

Algunos ejemplos de la información guardada son:

- Constantes: tipo, valor.
- Variables: tipo, dirección en memoria, tamaño.
- Funciones: número y tipo de los argumentos, tipo devuelto, dirección.

Es importante tener en cuenta que la información asociada con un identificador puede variar a lo largo del programa. Así, por ejemplo, el identificador `i` se refiere a dos variables distintas en el siguiente código:

```
int main() {
  int i=1;
  { float i=2.0;
    printf("%f\n", i); /* i es un float */
  }
  printf("%d\n", i); /* i es un int */
}
```

Una cuestión de gran importancia será encontrar una estructura de datos eficiente para acceder a los elementos de la tabla.

2.3.2. Gestión de errores

Es un hecho que al programar se cometen errores. Es más, se puede decir casi con total seguridad que un compilador encuentra más programas erróneos que correctos. Así pues, es importante que el compilador ayude en la detección y corrección de errores. Para ello, el compilador debe, ante un error:

- Diagnosticarlo de la manera más clara posible.
- Detener la generación de código.
- Intentar recuperarse para poder continuar el análisis.

Veremos que, en general, no es posible detectar los errores, sólo sus síntomas. Por ejemplo, ante la entrada `a:= b 1;` no es posible saber si falta un `+` entre la `b` y el `1` o si sobra un espacio o si sobra el uno... Sin embargo, será posible asegurar que la presencia de un error ha sido detectada lo antes posible; en nuestro ejemplo, detectaremos el error al ver el `1`.

2.4. Módulos externos al compilador

Aunque con lo que hemos visto hasta ahora podríamos considerar que tenemos el compilador completo, hay otros módulos que también se utilizan en el proceso de construcción de programas y que en muchos casos son programas independientes invocados por el propio compilador. Comentaremos brevemente tres: el preprocesador, en enlazador y el soporte en ejecución.

2.4.1. El preprocesador

En algunos lenguajes (probablemente el más conocido es C), existe una fase anterior al análisis léxico: el preproceso. En esta fase se realizan acciones tales como expansión de macros o inclusión de ficheros. Es interesante darse cuenta de que el preprocesador se comporta como un compilador muy restringido en sus capacidades y que tiene sus propias fases de análisis y síntesis.

2.4.2. El enlazador

En muchos casos, el resultado de la compilación no es un programa completo sino una parte de él: un fichero objeto. Una vez se han compilado todas las partes del programa, hay que unirlas, probablemente también empleando alguna biblioteca, para crear el programa final. El programa encargado de esto es el enlazador (*linker* en inglés).

2.4.3. Soporte en ejecución

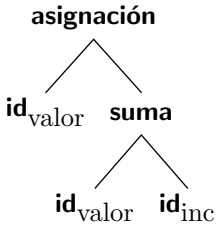
Además de las acciones especificadas por el programador, el código final tiene que hacer una serie de acciones necesarias para el buen funcionamiento del programa. La parte añadida al programa para hacer esto es lo que se conoce como soporte en ejecución. Algunas de sus funciones son preparar la memoria al comienzo de la ejecución, gestionar la memoria o la pila durante la ejecución y preparar la finalización de la ejecución de una manera razonable. Según los casos, este código puede estar contenido en funciones de la biblioteca estándar del lenguaje, puede ser añadido al programa o generado automáticamente.

3. La interpretación

Mientras que el objetivo de los compiladores es obtener una traducción del programa fuente a otro lenguaje, los intérpretes tienen como objeto la obtención de los resultados del programa. Para ello deben realizar dos tareas: analizar su entrada y llevar a cabo las acciones especificadas por ella.

La parte de análisis puede realizarse de manera idéntica a como se lleva a cabo en los compiladores. Es la parte de síntesis la que se diferencia sustancialmente. En el caso de la interpretación, se parte del árbol de sintaxis abstracta y se recorre, junto con los datos de entrada, para obtener los resultados.

En el caso del árbol



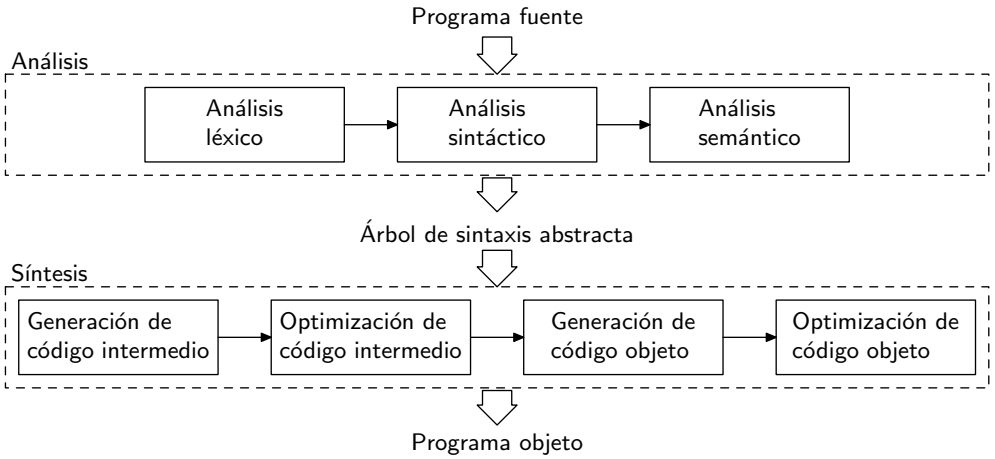
El recorrido consistiría en:

- Analizar el nodo asignación.
- Visitar su hijo derecho (la suma) para obtener el valor que hay que asignar:
  - Visitar el hijo izquierdo de la suma, recuperar el valor actual de `valor`.
  - Visitar el hijo derecho de la suma, recuperar el valor actual de `inc`.
  - Hacer la suma.
- Guardar el resultado de la suma en `valor`.

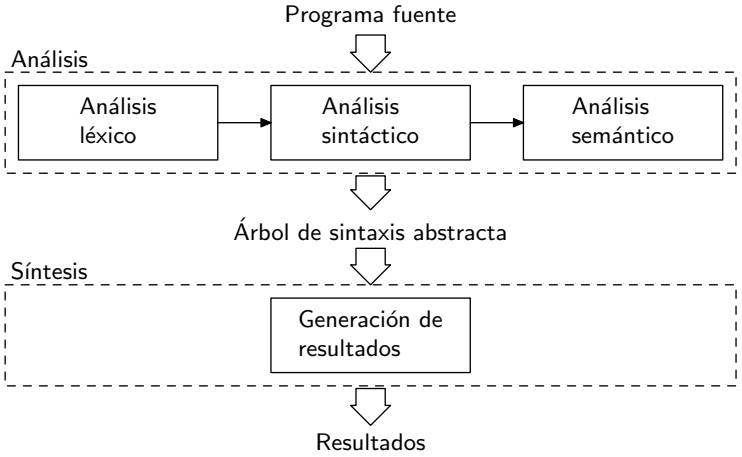
Actualmente es habitual encontrar híbridos entre la compilación y la interpretación que consisten en compilar a un lenguaje intermedio para una *máquina virtual* y después interpretar este lenguaje. Esta aproximación es la que se sigue, por ejemplo, en Java, Python o la plataforma .NET.

4. La arquitectura real de compiladores e intérpretes

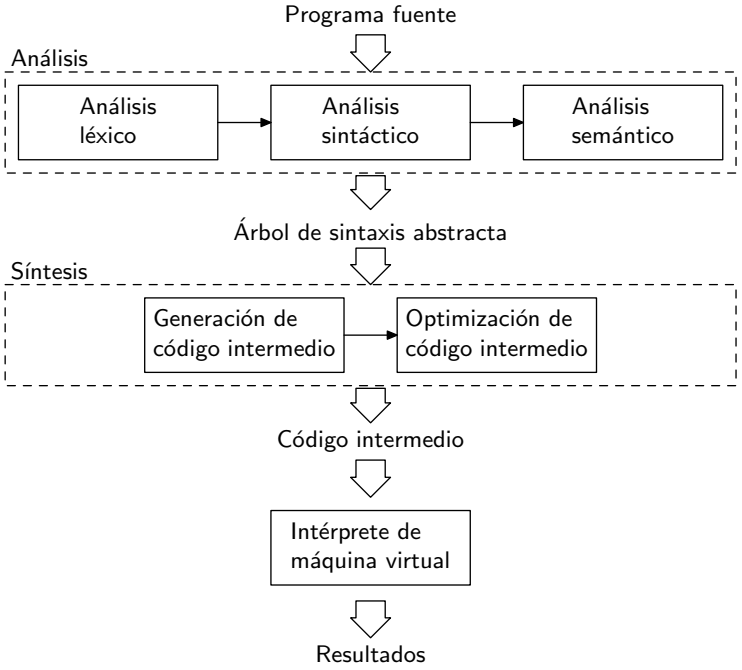
Siguiendo la descripción de las fases hecha más arriba, vemos que la estructura de un compilador es similar a la siguiente:



De manera análoga, un intérprete tendría esta estructura:



O, si utilizáramos una máquina virtual, el intérprete tendría esta estructura:



Sin embargo, en la práctica, la separación entre las distintas fases no está tan marcada. Lo habitual es que el analizador sintáctico haga las veces de "maestro de ceremonias", pidiendo al analizador léxico los componentos léxicos a medida que los va necesitando y pasando al analizador semántico la información que va obteniendo. De hecho, lo más normal es que este último (el analizador semántico) no exista como un módulo separado sino que esté integrado en el sintáctico. Así se elimina la necesidad de crear un árbol de análisis. Esta organización suele llamarse *traducción dirigida por la sintaxis*.

Si la memoria disponible es escasa, puede resultar imposible mantener una representación de todo el programa. En estos casos, se unen las fases de análisis y de síntesis. De esta manera, se va generando código al mismo tiempo que se analiza el programa fuente. Esta forma de trabajar era habitual en compiladores más antiguos y exige ciertas características especiales al lenguaje, como las declaraciones "forward" de Pascal.

5. Resumen del tema

- Dos etapas en la traducción: análisis y síntesis.
- Análisis:
  - Léxico: de caracteres a componentes.
  - Sintáctico: de componentes a árboles de análisis.
  - Semántico: de árboles de análisis a AST.
- Síntesis:
  - En compilación:
    - Generación de código intermedio.
    - Generación de código objeto.
    - Optimización (mezclada con las anteriores).
  - En interpretación, dos opciones:
    - Generación directa de resultados.
    - Generación de código intermedio e interpretación del código intermedio.