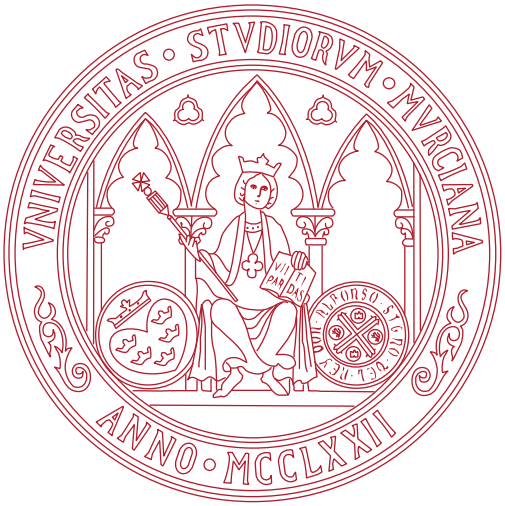


UNIVERSIDAD DE MURCIA

GRADO EN INGENIERÍA INFORMÁTICA

Apuntes de Compiladores

Autores:
María Antonia CÁRDENAS VIEDMA
mariancv@um.es
Eduardo MARTÍNEZ GRACIÁ
edumart@um.es
María Antonia MARTÍNEZ CARRERAS
amart@um.es

31 de enero de 2020

BIBLIOGRAFÍA

[1] J. G. Brookshear. *Teoría de la computación. Lenguajes formales, autómatas y complejidad*. Addison-Wesley Iberoamericana, 1993.

[2] Daniel Sánchez Álvarez, María Antonia Cárdenas Viedma, Juan Antonio Botía Blaya. *Traductores*. ICE. Universidad de Murcia, 2001.

[3] David A. Watt, Deryck F. Brown. *Programming language processors in Java*. Prentice Hall, 2000.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Pearson International Edition, second edition, 2007.

Estos apuntes no deben constituir la única fuente de estudio de la asignatura. Es muy recomendable que el alumno consulte la bibliografía, y en especial el excelente libro de Aho, Sethi y Ullman [4].

TEMA 1

TRADUCTORES E INTÉRPRETES

Índice

En este tema se justifica la construcción de compiladores con el objetivo de facilitar y dar mayor potencia a la tarea de programar ordenadores. Este objetivo se enfoca desde el punto de vista de la evolución de los lenguajes de programación. En este tema se introducen las principales nociones relativas al procesamiento de lenguajes formales: traductores, compiladores, intérpretes, lenguajes fuente y destino, y máquinas reales y abstractas. También se especifica y define la estructura típica de un compilador, describiendo cada una de las fases en las que se divide el proceso de compilación. Finalmente, se estudian formas interesantes de usar los procesadores de lenguajes: compiladores interpretados, compiladores portables, y las estrategias más comunes para poner en marcha el desarrollo de los propios compiladores.

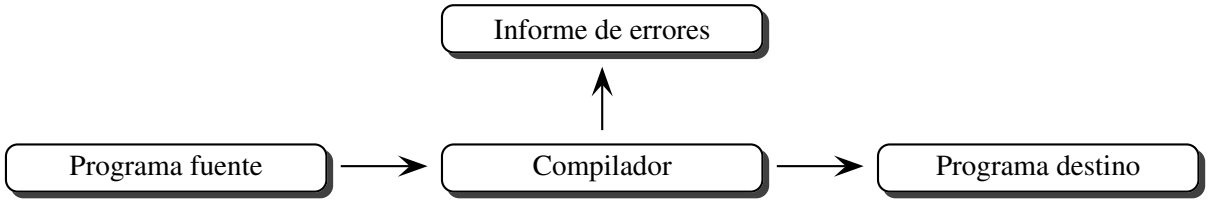
1.1. Origen de los compiladores

Los lenguajes de programación son herramientas básicas para los programadores. Un lenguaje de programación es una notación formal para expresar algoritmos. Los algoritmos manejan conceptos abstractos que son independientes de la notación concreta en la que se quieran expresar. Sin embargo, sin una notación formal no es posible compartir estos algoritmos ni razonar acerca de su validez.

Los programadores no sólo se preocupan por la forma de expresar y analizar los algoritmos, sino también por el modo de construir programas que los ejecuten. Por ello, los programadores necesitan facilidades para introducir dichos algoritmos en los ordenadores.

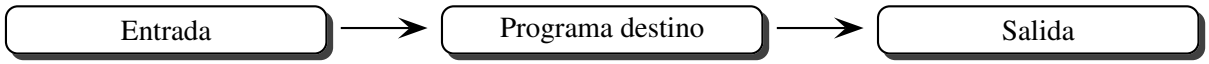
Los ordenadores ejecutan programas expresados en lenguaje máquina. Obviamente, los programas escritos en lenguaje máquina son extremadamente difíciles de leer, escribir, modificar y corregir, además de presentar una total dependencia de la máquina en la que se deben ejecutar. Los lenguajes de alto nivel, con capacidades expresivas cercanas al modo en que conceptualizamos los algoritmos, convierten la programación de ordenadores en una tarea mucho más sencilla. Sin embargo, los ordenadores siguen entendiendo únicamente el lenguaje máquina. Por esta razón, se necesita una herramienta que traduzca desde el lenguaje de alto nivel al lenguaje máquina. Esta herramienta es, precisamente, **el compilador**.

Un **compilador**, por tanto, traduce un programa escrito en lenguaje de alto nivel (*lenguaje fuente*) en un programa equivalente escrito en algún lenguaje de bajo nivel (*lenguaje destino*):

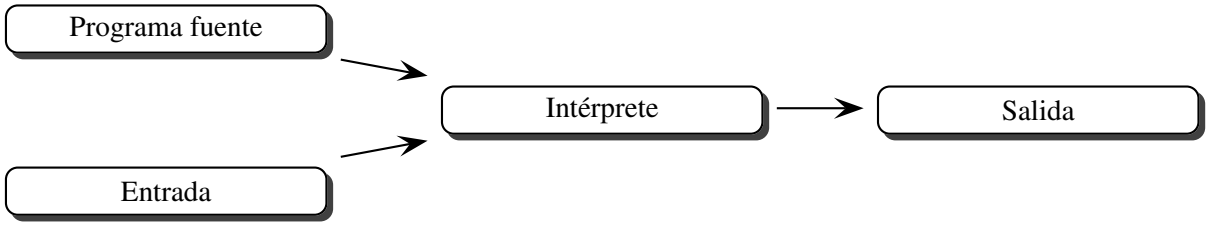


Otra función importante del compilador es **informar de cualquier error** que se detecte en el programa fuente durante el proceso de traducción.

Una vez obtenido el programa destino, si está escrito en lenguaje máquina se puede ejecutar:



Otro tipo común de procesador de lenguaje es el **intérprete**, que no genera un *programa destino* como resultado de una traducción, sino que ejecuta directamente las operaciones especificadas en el *programa fuente* con las entradas proporcionadas por el usuario:



El programa destino generado por un compilador produce una salida para una determinada entrada a mayor velocidad que un intérprete equivalente. Sin embargo, un intérprete puede producir un diagnóstico de errores mejor que un compilador, porque ejecuta el programa fuente instrucción a instrucción. Por otra parte, un intérprete aísla el entorno de ejecución de la máquina sobre la que se ejecuta, facilitando la posibilidad de ejecutar el programa fuente en múltiples plataformas (portabilidad).

Como dato anecdótico, se atribuye a menudo a Grace Murray Hopper la invención del término *compilador*. Se considera que esta científica fue una de las pioneras en el desarrollo de los lenguajes de programación. Entendió que la implementación de un lenguaje de alto nivel era similar a una "compilación[1] de una secuencia de subrutinas de una librería".

Nuestro conocimiento sobre cómo organizar y escribir compiladores ha aumentado mucho desde que comenzaron a aparecer los primeros compiladores a principios de los años cincuenta. Es difícil dar una fecha exacta de la aparición del primer compilador, porque en un principio gran parte del trabajo de experimentación y aplicación se realizó de manera independiente por varios grupos. Muchos de los trabajos inicialmente dedicados a la compilación estaban relacionados con la traducción de fórmulas aritméticas a código máquina.

En la década de 1950, se consideró a los compiladores como programas notablemente difíciles de escribir. El primer compilador de FORTRAN, por ejemplo, necesitó para su implementación 18 años-persona de trabajo[2]. Desde entonces se han descubierto técnicas sistemáticas para manejar muchas de las importantes tareas que surgen en la compilación. También se han desarrollado buenos lenguajes de implementación, entornos de programación y herramientas de software. Con estos avances, puede construirse un compilador real como proyecto de estudio en una asignatura sobre diseño de compiladores.

1.2. Evolución de los lenguajes de programación

Los primeros ordenadores se programaban usando los denominados *lenguajes de primera generación*. Estos lenguajes operan a nivel de código binario de la máquina, que consiste en una secuencia de ceros y unos que codifican explícitamente las operaciones y operandos, y el orden de ejecución. Las operaciones son de bajo nivel: mover datos de una localización a otra, sumar los contenidos de dos registros, comparar valores, y similares. No hace falta decir que el desarrollo de programas con este tipo de lenguajes es lento, tedioso y tendente a cometer fallos, y que los programas son difíciles de entender y modificar.

En torno al comienzo de los años 1950 se dio un primer paso hacia una programación más cómoda con los *lenguajes de segunda generación*. Estos lenguajes, llamados habitualmente *ensambladores*, permiten usar abreviaturas mnemotécnicas como nombres simbólicos que representan a las instrucciones de la máquina, y la abstracción cambia del nivel de puertas lógicas al de registros. El uso de códigos octales o hexadecimales se hace habitual. También se observan en estos lenguajes los primeros pasos hacia la estructuración de programas. Por ejemplo, con ellos es posible crear macros para que el programador pueda definir secuencias de instrucciones parametrizadas de uso frecuente.

Para hacer la tarea de la programación más sencilla, natural y robusta, se crearon los *lenguajes de tercera generación*, o lenguajes de alto nivel. Con ellos se pueden usar estructuras de control basadas en objetos de datos lógicos. Ofrecen un nivel de abstracción que permite la especificación de los datos, funciones o procesos y su control de forma independiente de la máquina. Entre estos lenguajes se encuentran Fortran, Cobol, C, C++ o Java. El diseño de programas para resolver problemas complejos es mucho más sencillo utilizando este tipo de lenguajes ya que se requieren menos conocimientos sobre la estructura interna del ordenador.

A la clasificación anterior se pueden añadir dos generaciones de lenguajes más. Los *lenguajes de cuarta generación* han sido diseñados para resolver problemas específicos, como SQL para las consultas de bases

[1] Compilar significa reunir informaciones procedentes de distintas fuentes.

[2] Es decir, el trabajo equivalente a 18 programadores dedicados a tiempo completo durante un año.

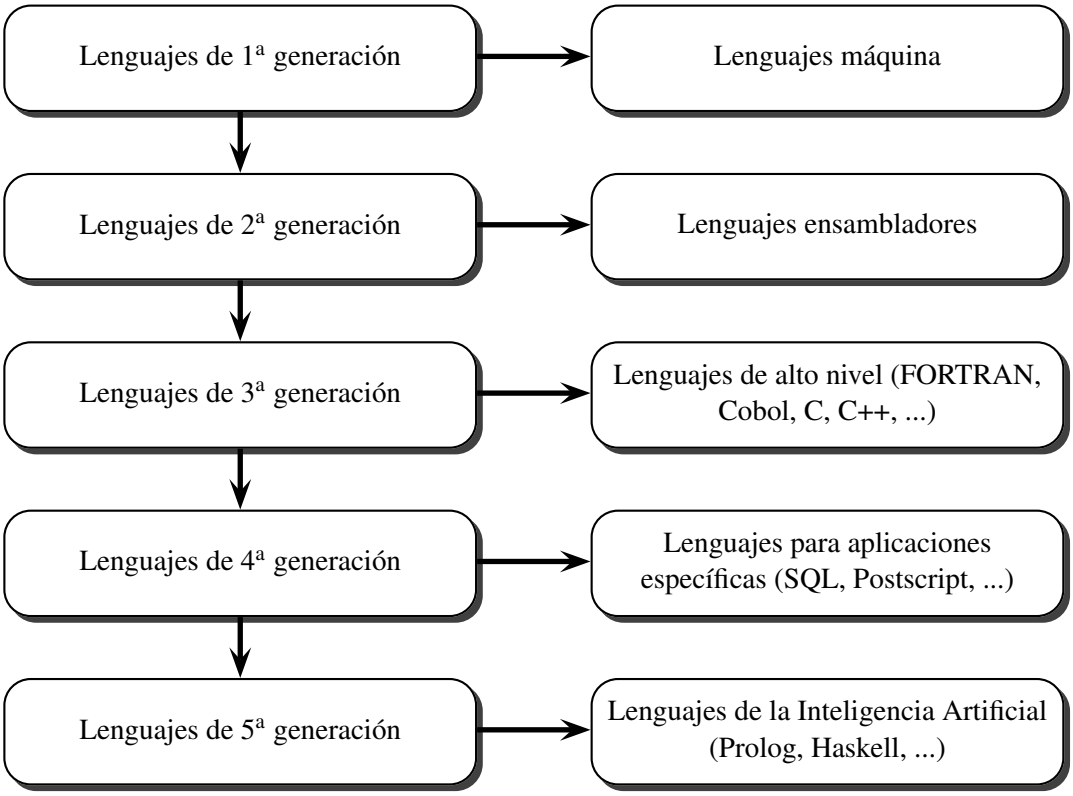


Figura 1.1: Evolución de los lenguajes de programación

de datos, Postscript para dar formato a textos o NOMAD para la generación de informes. Se suelen denominar *lenguajes de quinta generación* a los usados en Inteligencia Artificial, como Prolog o Haskell. Estos lenguajes también reciben el nombre de *lenguajes declarativos*, ya que con ellos se especifica *qué* cálculo se quiere realizar, mientras que los lenguajes tipo C, C++ o Java se denominan *lenguajes imperativos* porque con ellos se especifica *cómo* debe realizarse el cálculo. La figura 1.1 resume esta clasificación de los lenguajes de programación.

1.3. Programas traductores

El tipo de funcionalidad implementada por un compilador se puede generalizar mediante el concepto de programa **traductor**. Un traductor es un programa que acepta cualquier texto expresado en un lenguaje (el lenguaje fuente del traductor) y genera un texto *semánticamente equivalente* expresado en otro lenguaje (su lenguaje destino).

En esta asignatura estamos interesados en la construcción de traductores de textos que son programas. El texto en lenguaje fuente se llama **programa fuente**, y el texto en lenguaje destino se llama **programa objeto**. Los compiladores son programas que manipulan programas. Es fácil observar que un compilador es un tipo específico de traductor. Un compilador traduce desde un lenguaje de alto nivel a otro lenguaje de bajo nivel. Generalmente un compilador produce varias instrucciones de la máquina por cada instrucción fuente.

Antes de realizar cualquier traducción, un compilador comprueba que el texto fuente sea un programa correcto del lenguaje fuente. En caso contrario genera un informe con los errores. Estas comprobaciones tienen en cuenta la *sintaxis y las restricciones contextuales* del lenguaje fuente. Suponiendo que el programa fuente es correcto, el compilador genera un programa objeto que es semánticamente equivalente al programa fuente, es decir, que tiene los efectos deseados cuando se ejecuta. La generación del programa objeto tiene en cuenta tanto la *semántica* del lenguaje fuente como la del lenguaje destino.

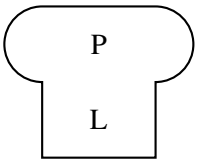
Un **ensamblador** es otro tipo de traductor. Su función es traducir desde un lenguaje ensamblador a su correspondiente código máquina. Por regla general, un ensamblador produce una instrucción de la máquina por cada instrucción fuente.

Los ensambladores y compiladores son las clases más importantes de traductores de lenguajes de programación, pero no son las únicas. A veces se utilizan los *traductores de alto nivel*, cuya fuente y destino son lenguajes de alto nivel. Otro caso son los *desensambladores*, que traducen un código máquina en su correspondiente lenguaje ensamblador. Un *descompilador* traduce un lenguaje de bajo nivel en un lenguaje de alto nivel.

1.4. Diagramas para representar a los programas traductores

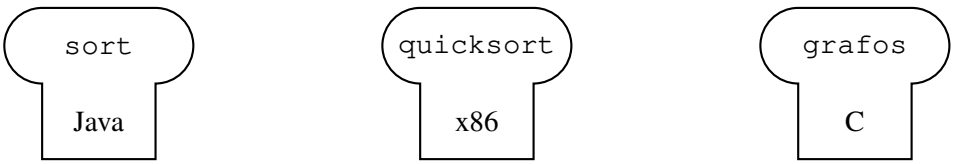
Además de los lenguajes fuente y destino, es evidente que en un programa traductor interviene un tercer lenguaje, que es el lenguaje de programación que se emplea para escribirlo. Dicho lenguaje se denomina *lenguaje de implementación*. Para evitar confusiones, es conveniente utilizar diagramas con los que se puedan representar de forma clara y sencilla a los programas corrientes y a los procesadores del lenguaje, reflejando los distintos tipos de lenguajes implicados. En esta asignatura se emplea el formato propuesto por David A. Watt y Deryck F. Brown, descrito en uno de los libros indicado en la bibliografía [3].

Un programa cualquiera se representa con un rótulo como el siguiente:



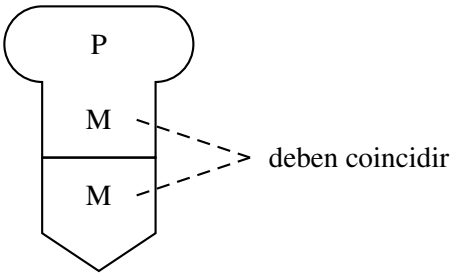
La cabeza del rótulo (parte superior) contiene el nombre del programa (P). La base del rótulo contiene el nombre del lenguaje de implementación (L), es decir, el lenguaje en el cual el programa está escrito.

A continuación se muestran los rótulos correspondientes a tres casos: un programa llamado `sort` escrito en Java, un programa llamado `quicksort` escrito en el código máquina de x86 (Intel) y un programa llamado `grafos` escrito en C:

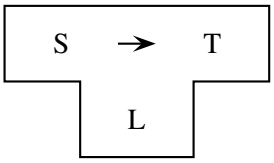


Los programas se ejecutan en máquinas. Una máquina que ejecuta código máquina M se representa por un pentágono dentro del cual se escribe M, como se muestra en el rótulo siguiente:

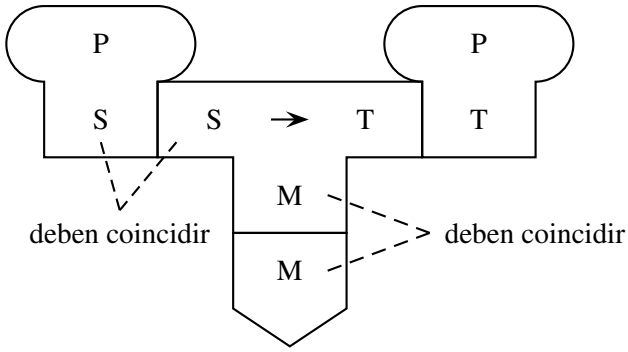
Un programa puede ejecutarse sobre una máquina sólo si se ha escrito en el código máquina apropiado. Consideremos que ejecutamos un programa P, escrito en código máquina M, sobre una máquina M. Este hecho se representa poniendo juntos el rótulo de P sobre el rótulo de M, como se muestra a continuación:



Por otro lado, un traductor se representa por un rótulo en forma de T. En la cabeza del rótulo del traductor aparecen los nombres del lenguaje fuente S y del lenguaje destino T, separados por una flecha. En la base del rótulo se escribe el lenguaje de implementación del traductor, L:



Un traductor S-en-T es, a su vez, un programa, y por tanto puede ejecutarse en una máquina M si su lenguaje de implementación es el código máquina de M. Cuando el traductor se ejecuta, traduce un programa fuente P, escrito en el lenguaje fuente S, a un programa objeto equivalente P, compuesto de instrucciones que pertenecen al lenguaje destino T. Todo esto queda reflejado en el siguiente diagrama:



En la figura 1.2 se muestra un diagrama que representa la compilación de un programa escrito en C. Utilizando un compilador C-en-x86, se traduce el programa fuente `sort` a un programa objeto equivalente, escrito en código máquina x86. Ya que el compilador está escrito en código máquina x86, el compilador puede ejecutarse sobre una máquina x86. La segunda parte del diagrama muestra el programa objeto ejecutándose sobre una máquina x86.

El comportamiento de un traductor puede resumirse en unas pocas y sencillas reglas:

- Un traductor (como cualquier otro programa) puede ejecutarse sobre una máquina M si y sólo si está escrito en código máquina M.
- El programa fuente debe escribirse en el lenguaje fuente S del traductor.
- El programa objeto generado estará escrito en el lenguaje destino T del traductor.
- El programa objeto generado debe ser semánticamente equivalente al programa fuente.

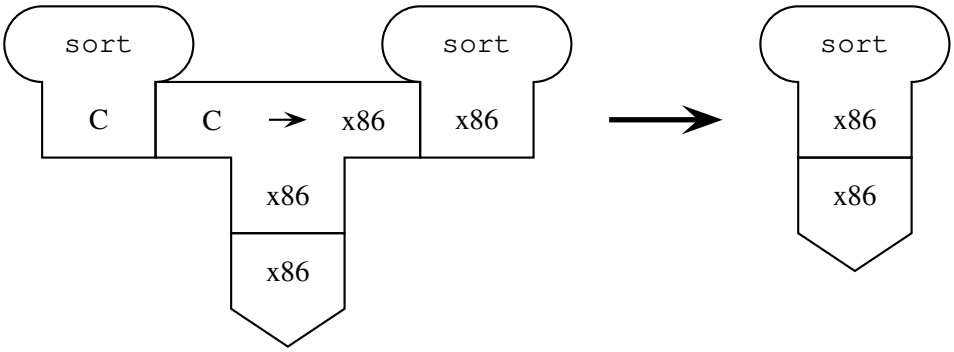
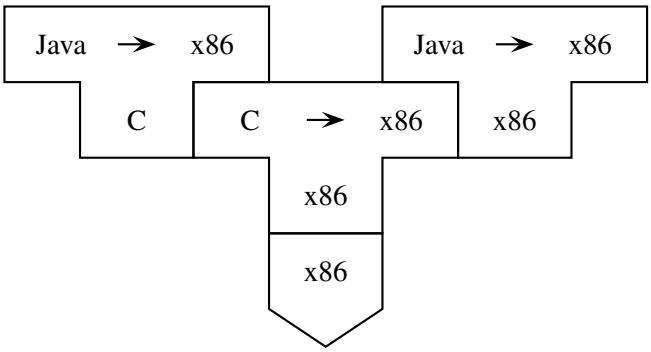


Figura 1.2: Compilación de un programa en C

1.5. Compiladores

Como ya se ha indicado, un compilador es, a su vez, un programa escrito en algún lenguaje. Por tanto, un compilador podría tratarse como programa fuente de otro compilador. Por ejemplo, supongamos que se tiene un compilador Java-en-x86 escrito en el lenguaje de implementación C. Como el lenguaje de implementación no es código máquina, este compilador no se puede ejecutar directamente. Sin embargo, se puede traducir por medio de otro compilador C-en-x86, como se muestra a continuación:



El programa objeto es un compilador Java-en-x86 escrito en código máquina x86. Este nuevo compilador podría ejecutarse para compilar programas Java. En general, todos los procesadores de lenguajes son programas, y como tales pueden ser manipulados por otros procesadores de lenguajes.

En ocasiones, puede ocurrir que el lenguaje de implementación y el lenguaje fuente del procesador *sean el mismo*. Por tanto, el procesador puede usarse para procesarse a sí mismo. Esta situación se denomina **arranque** (bootstrapping). Aunque la idea parezca una paradoja, puede resultar útil en varios escenarios. El primer compilador capaz de compilar su propio código fuente fue el creado para Lisp por Hart y Levin en el MIT en 1962. Desde 1970 se ha convertido en una práctica común escribir el compilador en el mismo lenguaje que éste compila, aunque Pascal y C han sido también alternativas muy usadas.

1.5.1. Contexto de un compilador

En el proceso de construcción de un programa suelen intervenir, aparte del compilador, otros programas traductores, como muestra la figura 1.3.

El *preprocesador* es un traductor cuyo lenguaje fuente es una forma extendida de algún lenguaje de alto nivel, y cuyo lenguaje objeto es la forma estándar del mismo lenguaje. Realiza la tarea de reunir el programa fuente, que a menudo se divide en módulos almacenados en archivos diferentes (ficheros de cabecera). También puede expandir abreviaturas, llamadas macros, en sentencias del lenguaje fuente. El programa

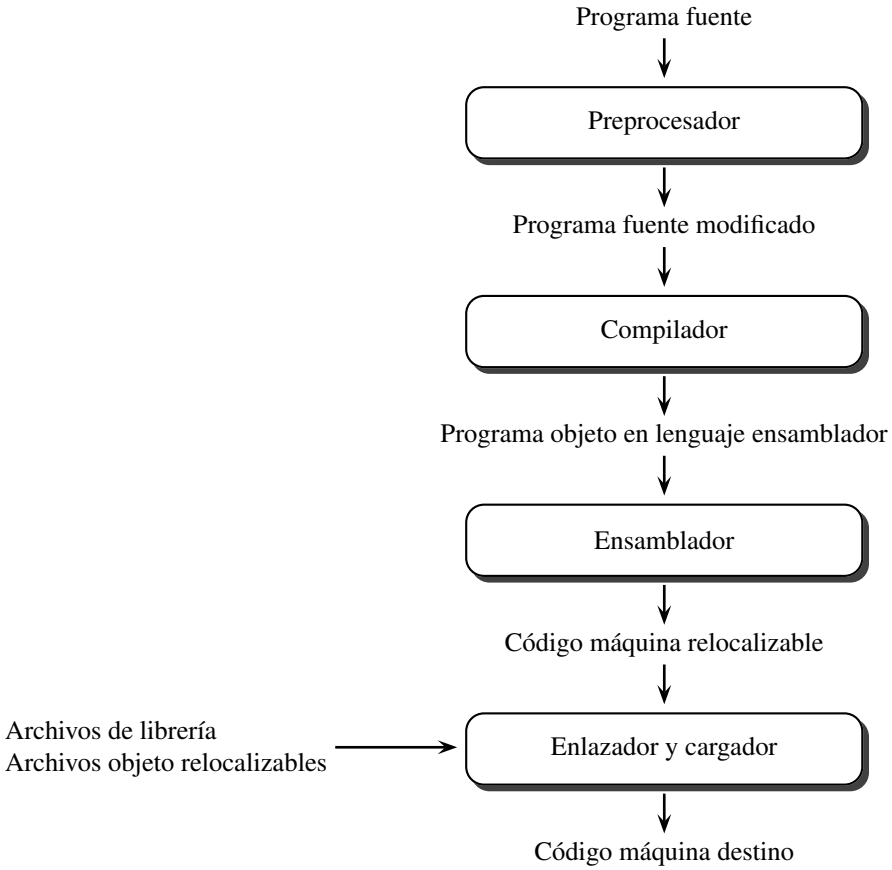


Figura 1.3: Contexto de un compilador

objeto producido por un preprocesador puede ser manejado por el procesador usual del lenguaje estándar.

Es frecuente encontrar compiladores que traducen de un lenguaje de alto nivel a lenguaje ensamblador. La razón es simple: implementar un compilador que genere una salida en lenguaje ensamblador es más sencillo que desarrollarlo para que genere directamente código máquina. Lógicamente, es necesario emplear un programa ensamblador que traduzca la salida del compilador a código máquina.

Normalmente, si se siguen unas buenas prácticas de desarrollo de software, nos encontraremos con programas que están divididos en varios ficheros fuente. La compilación de cada uno de ellos produce un fichero objeto en código máquina *relocalizable*, es decir, código máquina con direcciones de memoria relativas (offsets) a una dirección base que representa el comienzo de la memoria usada por el fragmento de código construido a partir de ese fichero fuente. En el caso de usar llamadas a funciones implementadas en otros ficheros, o en librerías, el código relocalizable incluirá referencias no definidas a direcciones externas que deberán resolverse posteriormente.

La función del *enlazador* es unir todos los ficheros de código máquina relocalizable, así como el código de las librerías usadas, ajustando adecuadamente las direcciones de memoria relativas, y resolviendo las direcciones externas de memoria. El resultado es una única unidad de código máquina lista para ejecutarse.

La tarea final de ejecución del programa generado por el enlazador la lleva a cabo el *cargador*. Generalmente se trata de una parte del sistema operativo[3] que realiza varias operaciones:

- Valida los permisos de ejecución del programa, y la disponibilidad de memoria.
- Copia el programa a memoria. En caso de usar librerías dinámicas, el cargador también las situará en memoria y resolverá las referencias a las funciones llamadas.

[3] En el caso de los sistemas tipo UNIX/Linux, el cargador se invoca con la llamada al sistema `execve()`.

- Copia los argumentos de línea de comandos a la pila.
- Inicializa los registros del procesador (por ejemplo, el puntero de la pila).
- Salta al punto de entrada del programa (función principal).

1.5.2. Fases de un compilador

En el proceso de compilación podemos distinguir dos tareas bien diferenciadas:

- *Análisis*: se determina la estructura y el significado de un código fuente. Esta parte del proceso de compilación divide al programa fuente en sus elementos componentes y crea una representación intermedia de él.
- *Síntesis*: a partir de esa representación intermedia se traduce el código fuente a un código de máquina equivalente. En esta etapa es necesario usar técnicas más especializadas que las manejadas durante el análisis.

Conceptualmente, un compilador opera en estas dos etapas, que a su vez se pueden dividir en varias fases. En la figura 1.4 se muestra la descomposición típica de un compilador. En la práctica, sin embargo, se pueden agrupar algunas de estas fases, de modo que los datos intermedios entre ellas pueden no ser construidos explícitamente.

Las cuatro primeras fases de la figura 1.4 conforman la parte de *análisis* del compilador, también llamada *front-end*. En esta parte se determina la estructura y el significado del programa fuente, con el objetivo de crear una representación intermedia del programa fuente, fácil de optimizar y de traducir a código dependiente de la máquina. Si la fase de análisis detecta que el programa fuente está mal formado desde el punto de vista sintáctico, o es incoherente desde el punto de vista semántico, el compilador debe informar mediante mensajes aclarativos, facilitando al usuario la corrección de dichos errores. Durante la etapa de análisis el compilador recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada *tabla de símbolos*, que junto con la representación intermedia se pasa al bloque de síntesis.

La parte de *síntesis* del compilador, también llamada *back-end*, construye el programa objeto destino a partir de la representación intermedia y de la información contenida en la tabla de símbolos.

Algunos compiladores tienen una fase de optimización del código independiente de la máquina, situada entre el front-end y el back-end. El propósito de esta fase es realizar transformaciones en la representación intermedia, de modo que el back-end pueda producir un mejor programa objeto. Como la optimización es opcional, cualquiera de las dos fases de optimización mostradas en la figura 1.4 pueden no estar presentes en un compilador.

1.5.2.1. Análisis léxico

La parte del compilador que realiza el análisis léxico se llama *analizador léxico*, *scanner* o *explorador*.

La tarea básica que realiza el analizador léxico es transformar un flujo de caracteres de entrada en una serie de componentes léxicos, también llamados *tokens*. Se encargaría, por tanto, de reconocer los identificadores de variables o funciones, palabras clave, constantes, operadores, etc., del programa fuente.

La secuencia de caracteres que forman el token se denomina *lexema*. No hay que confundir el concepto de token con el de lexema. A un mismo token le pueden corresponder varios lexemas. Por ejemplo, se pueden reconocer como tokens de tipo identificador a todas las instancias de variables de un programa.

Aunque para analizar sintácticamente una expresión sólo nos hará falta el código del token, el lexema debe ser recordado para usarlo en fases posteriores dentro del proceso de compilación. El analizador léxico es el único componente del compilador que tendrá acceso al programa fuente. Por tanto, debe de encargarse de proporcionar dos datos sobre cada token al analizador sintáctico: el código de token reconocido

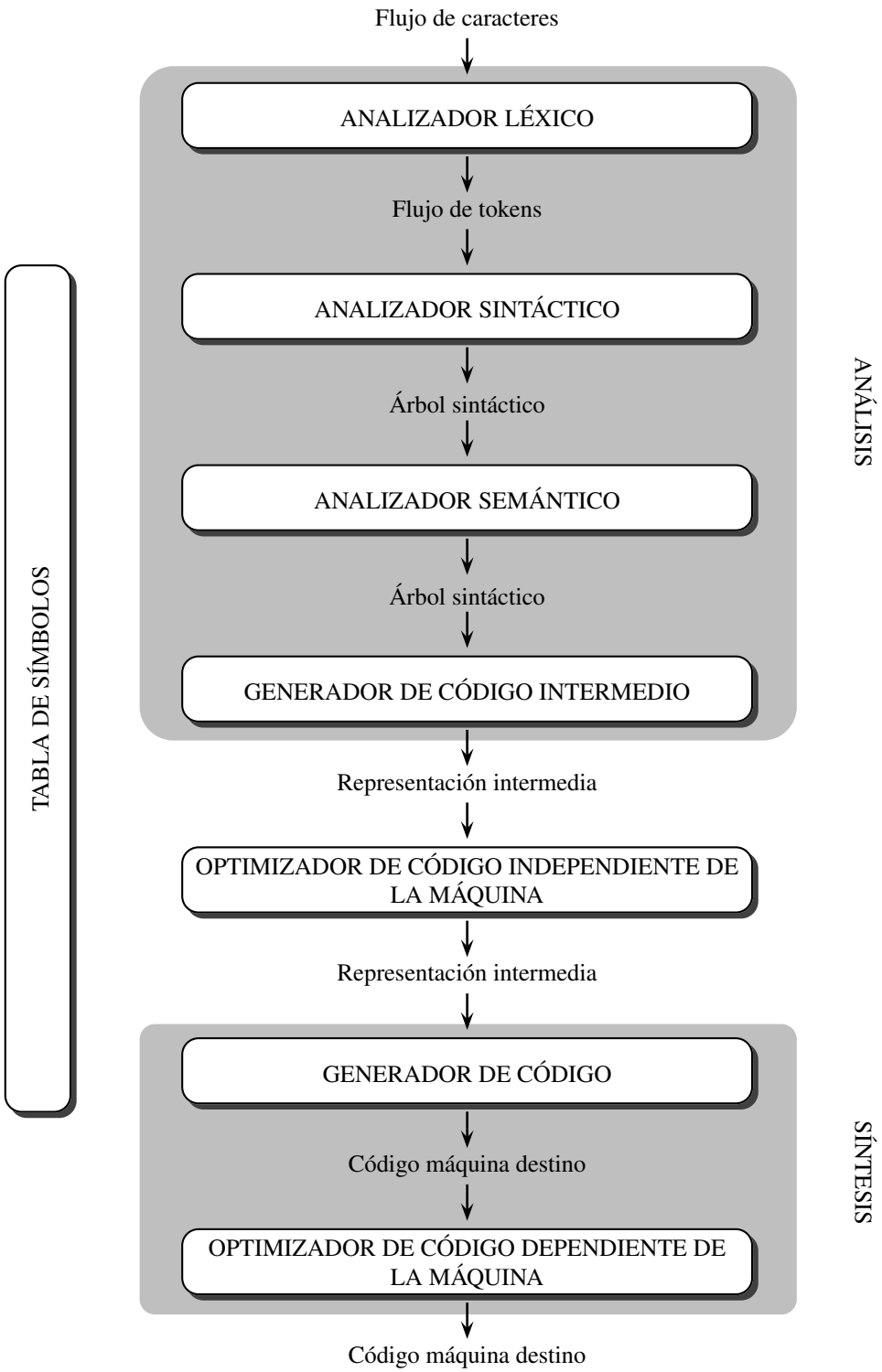


Figura 1.4: Fases de un compilador

y la información del lugar en el que está almacenado el lexema del token, en caso de que sea necesario. Representaremos este par de datos de la forma:

```
< código-token, puntero-a-lexema >
```

Existen varias alternativas para realizar el interfaz entre el analizador léxico y el sintáctico. Una primera posibilidad es que el analizador léxico sitúe los lexemas reconocidos en entradas de la tabla de símbolos. En este caso, `puntero-a-lexema` podría ser el índice de la entrada en la tabla de símbolos que almacena dicho lexema. Otras fases posteriores del compilador pueden completar la información sobre cada dato almacenado en la tabla de símbolos.

Otra posibilidad es que el analizador léxico no actualice la tabla de símbolos. En este caso pasaría una copia del lexema al analizador sintáctico de modo que `puntero-a-lexema` sería un puntero al comienzo de la cadena de caracteres del lexema. El analizador sintáctico sería el encargado de introducir este dato en la tabla de símbolos. Esta opción es mejor en el caso de que el tipo de la entrada en la tabla de símbolos dependa de la función sintáctica del token (muchos lenguajes emplean identificadores similares para representar variables y funciones).

Supongamos que desarrollamos un compilador para un lenguaje que permite expresar asignaciones de expresiones aritméticas a variables. La siguiente podría ser una asignación válida:

```
posicion = inicial + velocidad * 60
```

Empleando la primera alternativa mencionada anteriormente, un analizador léxico podría agrupar los caracteres de esta asignación en los siguientes pares:

```
<IDENT,1> <ASIG,> <IDENT,2> <SUMA,> <IDENT,3> <MULT,> <NUM,60>
```

quedando la tabla de símbolos con la siguiente información:

1	`posicion`	···
2	`inicial`	···
3	`velocidad`	···

Es interesante realizar algunas apreciaciones en este ejemplo. Los códigos de los tokens (`IDENT`, `ASIG`, `SUMA`, `MULT` y `NUM`) pueden considerarse constantes o valores de tipo enumerado que forman parte de la definición del interfaz entre el analizador léxico y el sintáctico. Por otra parte, algunos tokens no aparecen acompañados de punteros a lexemas. Es el caso de los tokens con código `ASIG`, `SUMA` y `MULT`. La razón de que no tengan información del lexema asociado es sencilla: estos tokens sólo pueden tener un único lexema, de modo que es irrelevante este dato. Finalmente, el último token, de tipo `NUM`, debería tener un apuntador a una entrada en la tabla de símbolos que albergase el lexema 60. Sin embargo, el lexema se muestra directamente en el par que representa al token. Aunque se tratarán con mayor amplitud en el tema 2 las cuestiones relativas al análisis léxico, cabe indicar aquí que no es habitual introducir los lexemas de los valores literales en la tabla de símbolos.

Los componentes léxicos de un lenguaje pueden ser generados por gramáticas regulares y, por tanto, reconocidos por autómatas finitos. Ésta es la herramienta fundamental para la implementación de los analizadores léxicos. Si durante la fase de análisis léxico se encuentra algún lexema que no corresponda con la expresión regular de ningún token válido, el analizador debe indicar un mensaje de *error léxico* e intentar recuperarse para continuar el análisis.

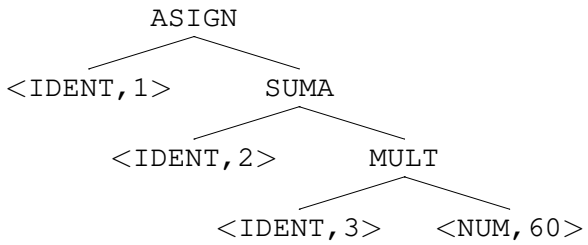
Puesto que el analizador léxico es el componente del compilador que actúa de interfaz con el código fuente, debe encargarse de eliminar los símbolos no significativos del programa, como espacios en blanco, tabuladores, comentarios, etc. Es conveniente también que esta fase quede separada de la siguiente por razones de eficiencia. Otra ventaja adicional de esta separación es que el lenguaje fuente puede tener diferentes representaciones (código de caracteres y expresiones regulares para representar a los componentes léxicos),

y únicamente sería necesario modificar el analizador léxico para poder tratarlas, sin tener que alterar el resto del compilador.

Actualmente se dispone de herramientas como *Flex* que permiten generar de forma automática un analizador léxico a partir de la especificación de las expresiones regulares de sus tokens, simplificando notablemente la construcción de esta fase del compilador.

1.5.2.2. Análisis sintáctico

La parte del compilador que realiza el análisis sintáctico se llama *analizador sintáctico* o *parser*. Empleando los códigos de los tokens producidos por el analizador léxico, el analizador sintáctico crea una representación intermedia del programa fuente que refleja su estructura gramatical. Esta representación suele tener forma de *árbol sintáctico*. Por ejemplo, la secuencia de tokens indicada en el apartado anterior se podría representar con el siguiente árbol:



El árbol tiene un nodo interior etiquetado con `MULT`, que indica explícitamente la primera operación que es necesario ejecutar para que la expresión aritmética se evalúe correctamente. Sus operandos son el valor del identificador `velocidad` y la constante numérica `60`. El resultado de esta operación se debe sumar seguidamente con el valor del identificador `inicial`, y finalmente se debe almacenar el resultado en la localización del identificador `posicion`.

De la forma de construir el árbol de análisis se derivan dos tipos de analizadores sintácticos:

- Cuando se parte del axioma de la gramática y se va construyendo el árbol de análisis hacia abajo, utilizando derivaciones por la izquierda, hasta llegar a los nodos hoja, se dice que el análisis es *descendente*.
- Por el contrario, cuando se parte de la cadena de entrada y se va generando el árbol hacia arriba mediante reducciones por la izquierda, hasta conseguir llegar al axioma de la gramática, se dice que el análisis es *ascendente*.

Si el programa no tiene una estructura sintáctica correcta, el analizador sintáctico no podrá encontrar un árbol de derivación y deberá generar mensajes de *error sintáctico*.

La división entre análisis léxico y sintáctico es algo arbitraria. Generalmente se elige una división que simplifique la tarea completa del análisis. Un factor para determinar cómo realizarla es comprobar si una construcción del lenguaje fuente es inherentemente recursiva o no. Las construcciones léxicas no requieren recursión, mientras que las sintácticas suelen requerirla.

Los *autómatas con pila* describen la forma de analizar lenguajes generados por *gramáticas libres de contexto*. Éstas formalizan la mayoría de las reglas recursivas que pueden usarse para guiar el análisis sintáctico. Sin embargo, es importante destacar que la mayor parte de los lenguajes de programación pertenecen realmente al grupo de los *lenguajes dependientes del contexto*.

Existen generadores automáticos de analizadores sintácticos para algunos subconjuntos de gramáticas libres de contexto. Uno de los más conocidos es *Bison*.

1.5.2.3. Análisis semántico

Para que la definición de un lenguaje de programación sea completa, aparte de las especificaciones de su sintaxis (estructura o forma en que se escribe un programa), necesitamos también especificar su semántica (significado o definición de lo que realmente hace un programa). La fase de análisis semántico de un compilador se encarga de tratar dos aspectos de la traducción: la verificación de las construcciones sintácticas que no pueden ser tratadas con gramáticas libres de contexto, y el cálculo de valores semánticos que garanticen la generación de código correcto para cada posible construcción del lenguaje fuente.

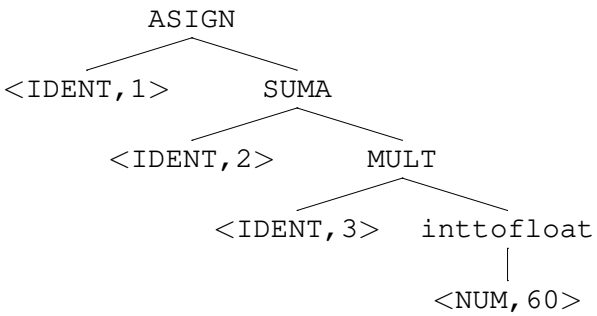
La sintaxis de un lenguaje de programación se suele dividir en componentes libres de contexto y sensibles al contexto. La *sintaxis libre de contexto* define secuencias legales de símbolos, independientemente de cualquier noción sobre el contexto o circunstancia particular en que aparecen dichos símbolos. Por ejemplo, una sintaxis libre de contexto puede informarnos de que la sentencia `A := B + C` es correcta, mientras que `A := B*` no lo es.

Sin embargo, no todos los aspectos de un lenguaje de programación pueden ser descritos mediante este tipo de sintaxis. Éste es el caso, por ejemplo, de las reglas de alcance para variables, o de la compatibilidad de tipos. Estos son *aspectos sensibles al contexto* de la sintaxis que define al lenguaje de programación. Por ejemplo, `A := B + C` podría no ser una sentencia correcta si las variables no están declaradas, o son de tipos incompatibles.

Teniendo en cuenta que en la mayoría de los casos se utilizan, por simplicidad, gramáticas libres de contexto para especificar la sintaxis de los lenguajes de programación, es necesario hacer un tratamiento especial para las restricciones sensibles al contexto. Éstas pasan a tratarse como parte de la semántica del lenguaje de programación.

Por tanto, la fase de análisis semántico revisa el programa fuente para tratar de encontrar *errores semánticos*, y reúne la información sobre los tipos de datos para la fase posterior de generación de código. Para esto se utiliza la representación intermedia que se construye en la fase de análisis sintáctico, así como la tabla de símbolos.

Una tarea importante a realizar en esta fase es la *verificación de tipos*. Esta tarea consiste en la verificación de que cada operador se aplica a operandos con tipos compatibles, de acuerdo con la especificación del lenguaje fuente. Muy frecuentemente esta especificación puede permitir ciertas *conversiones de tipos* en los operandos. Por ejemplo, supongamos que los identificadores `posicion`, `inicial` y `velocidad` del ejemplo indicado en el apartado anterior han sido declarados como números en punto flotante. La verificación de tipos del analizador semántico puede convertir en un punto flotante la constante numérica entera `60`, añadiendo el operador `inttofloat`, que explícitamente convierte el entero que se le pasa como argumento en un número en punto flotante:



De esta forma, la operación de multiplicación se realizará entre dos datos del mismo tipo.

Resumiendo, algunas de las comprobaciones que se realizan en el análisis semántico son:

- Chequeo y conversión de tipos.

- Comprobación de que el tipo y número de parámetros en la declaración de funciones coincide con los de las llamadas a esa función.
- Comprobación del rango para índices de arrays.
- Comprobación de la declaración de variables.
- Comprobación de las reglas de alcance de variables

Es muy importante la utilización de **métodos formales** para especificar la semántica de un lenguaje de programación. En los comienzos del desarrollo de los compiladores se utilizó el lenguaje natural para realizar esta especificación, lo que provocó la aparición de múltiples errores y ambigüedades en las definiciones de los lenguajes. Esto, a su vez, provocó incompatibilidades entre distintas implementaciones de un mismo lenguaje.

Existen diferentes aproximaciones para la especificación formal de la semántica. Damos a continuación una breve descripción de algunos de los métodos formales para realizarla.

- La **semántica operacional** define una implementación del lenguaje, a partir de la cual las demás implementaciones deberían ser modeladas. Esta implementación suele estar basada en una máquina abstracta idealizada.

  Esta especificación de la semántica describe cómo cada construcción del lenguaje altera el estado de la máquina abstracta. De esta forma es posible deducir la acción de cualquier programa concreto, dado un conjunto específico de datos de entrada.

  A primera vista, la aproximación operacional parece muy atractiva: es conceptualmente fácil de comprender y la flexibilidad en el diseño de la máquina abstracta puede simplificar la tarea de la especificación. Sin embargo, esta técnica tiene un uso práctico limitado, porque no permite realizar aserciones abstractas sobre la semántica de programas en general. Por otro lado, suele ser muy difícil, o incluso imposible, relacionar la implementación de la máquina abstracta con la de un hardware real.

- La **semántica denotacional** es una técnica de especificación que no se basa en la descripción de una implementación del lenguaje. Por el contrario, esta técnica equipara las construcciones del lenguaje a objetos matemáticos, como números, conjuntos o funciones. Combinando las distintas construcciones existentes en un programa se puede llegar a un objeto matemático que representa el significado del programa completo.

  Este método permite hacer declaraciones generales sobre programas, aunque con cierta dificultad, ya que las matemáticas implicadas son especializadas y pueden resultar complejas.

  Esta técnica es más útil para el diseñador del lenguaje que para el implementador.

- La **aproximación axiomática** es una técnica que se basa en la especificación de axiomas y reglas de inferencia de la lógica simbólica para definir de qué forma transforma los datos cada construcción del lenguaje.

  Dado un programa, es posible deducir propiedades sobre el mismo usando demostraciones formales que manejan los axiomas y las reglas establecidas en la definición del lenguaje.

  Las definiciones axiomáticas son más abstractas que las denotacionales y operacionales, y las propiedades que se pueden evaluar sobre un programa no siempre son suficientes para determinar su significado completo.

- Finalmente, las **gramáticas atribuidas** permiten realizar comprobaciones semánticas a partir del árbol de derivación que proporciona el analizador sintáctico. La gramática se amplía con ciertos atributos y con predicados que permiten evaluar sus valores. Este es el tipo de técnica que se emplea en el tema 4 de esta asignatura.

1.5.2.4. Generación de código intermedio

En el proceso de traducción de un programa fuente a código máquina, un compilador puede construir una o más representaciones intermedias, con una gran variedad de formas. Los árboles sintácticos usados en el análisis sintáctico y semántico son un ejemplo de representación intermedia.

Tras las etapas de análisis sintáctico y semántico del programa fuente, muchos compiladores generan una representación intermedia explícita en un lenguaje de bajo nivel que puede considerarse el código de una máquina abstracta. Esta representación intermedia debe cumplir dos propiedades:

- Debe ser fácil de producir.
- Debe ser fácil de traducir a código objeto.

En el tema 5 se estudian distintas formas usadas para esta representación intermedia. Una de ellas es el *código de tres direcciones*, que consiste en una secuencia de instrucciones similar al ensamblador con un máximo de tres operandos por instrucción. Cada operando actúa como un registro. Por ejemplo, el código intermedio del árbol sintáctico indicado en el apartado anterior puede constar de la siguiente secuencia de operaciones:

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Se pueden indicar varios aspectos interesantes respecto al código de tres direcciones. En primer lugar, las instrucciones de asignación tienen, como mucho, un operador en el lado derecho. Por tanto, estas instrucciones fijan el orden en que se deben realizar las operaciones (la multiplicación precede a la suma). En segundo lugar, el compilador debe generar un nombre de variable temporal para almacenar el valor calculado por cada operación. En tercer lugar, algunas instrucciones pueden tener menos de tres operandos (como la primera y la cuarta).

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

1.5.2.5. Optimización de código

Las fases de optimización de código tratan de mejorar bien el código intermedio o bien el código objeto, de modo que el programa traducido tenga un rendimiento mejor. Evidentemente, esta transformación debe dar como resultado un código semánticamente equivalente. La optimización puede realizarse en función de varios parámetros:

- Reducción del consumo de memoria.
- Reducción del consumo de energía del procesador.
- Aumento de la velocidad de ejecución de un programa. Éste es el tipo de optimización más importante.

La generación de código óptimo es un problema NP-completo. Por tanto, en realidad la fase de optimización no suele producir el código óptimo, sino un código mejorado.

Hay mucha variación en la cantidad de mejora de código obtenida por los distintos compiladores. Los que realizan un esfuerzo más exhaustivo en la optimización, denominados *compiladores optimizadores*, dedican una parte significativa del tiempo de compilación en esta tarea. Sin embargo, hay optimizaciones sencillas que mejoran sensiblemente el tiempo de ejecución del programa objeto sin necesidad de retardar demasiado la compilación.

A causa del tiempo requerido en esta fase, es muy usual que el compilador ofrezca al usuario la posibilidad de desactivar la opción de optimización del generador de código durante la fase de desarrollo o

depuración de programas.

En general, las técnicas de optimización se basan en un análisis de la estructura del programa y del flujo de datos. Es muy usual dividir el programa en regiones para realizar una parte de esta tarea. Tras una etapa de optimización local, se pasa a realizar optimizaciones a nivel global.

Por ejemplo, el código intermedio indicado en el apartado anterior puede ser objeto de varias optimizaciones. En primer lugar, un optimizador puede deducir que la conversión de la constante numérica entera 60 a un valor en punto flotante puede realizarse una vez en tiempo de compilación, de modo que la operación `inttofloat` puede ser eliminada, reemplazando `60` por la constante en punto flotante `60.0`. Además, `t3` se usa sólo una vez para pasar su valor a `id1`, de modo que puede eliminarse. El resultado de la optimización puede ser la secuencia reducida:

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Por otra parte, las técnicas de optimización pueden ser de dos tipos:

- *Independientes de la máquina*, si no tienen en cuenta la estructura del hardware ni el conjunto de instrucciones de la máquina destino. Por tanto, tienen un carácter general y se suelen aplicar al código intermedio.
- *Dependientes de la máquina*, en el caso de que tomen en consideración el hardware y el conjunto de instrucciones, ya que tienen en cuenta criterios como la asignación de registros o la selección de instrucciones.

1.5.2.6. Generación de código

La fase final del compilador es la generación de código objeto partiendo de una representación intermedia del programa fuente.

Existen diferentes métodos para realizar la generación de código. Cuando el compilador afronta esta etapa usando el árbol de análisis de la entrada, se emplea un método conocido como *traducción dirigida por la sintaxis*. Se trata, por tanto, de una técnica de traducción basada en la gramática del lenguaje. Por otra parte, si la etapa de generación del código objeto tiene como entrada código intermedio, sólo es necesario realizar una traducción relativamente sencilla de éste al código objeto.

En cualquier caso, existen varias posibilidades en cuanto al formato que puede tener el código objeto:

- Generar directamente código máquina. En este caso debe resolverse, entre otras cuestiones, el problema de la reserva de memoria para los identificadores que aparezcan en el programa. Esto hace necesario construir un mapa de direcciones que asocie a cada identificador su correspondiente dirección de memoria o registros del procesador.
- Generar código en lenguaje ensamblador de la máquina destino. Posteriormente habría que traducirlo mediante un programa ensamblador a código objeto relocalizable.

  Esta forma de generar código es más sencilla y permite compilar por separado distintos programas que pueden interactuar entre sí, usando librerías de rutinas. De hecho, esta técnica es muy común en compiladores que trabajan bajo entorno tipo UNIX.

Por ejemplo, el código generado partiendo del código intermedio indicado en el apartado anterior podría ser el siguiente:

```
LDF   R2,   id3
MULF  R2,   R2,  #60.0
LDF   R1,   id2
ADDF  R1,   R1,  R2
STF   id1,  R1
```

Las operaciones que manipulan números en punto flotante terminan con la letra `F`. El primer operando de cada instrucción especifica un destino. El código comienza cargando la dirección `id3` en el registro `R2`. A continuación multiplica el registro por `60.0`. El carácter `#` significa que `60.0` debe tratarse como una constante. La tercera instrucción mueve `id2` al registro `R1` y la cuarta añade a este registro el valor previamente calculado en el registro `R2`. Finalmente, el valor del registro `R1` se almacena en la dirección `id1`.

En cualquier caso, la generación de código es una tarea complicada, que requiere profundos conocimientos del hardware de la máquina destino, con el fin de aprovechar al máximo los recursos de la misma para que el código final resulte lo más eficiente posible.

1.5.3. Bases gramaticales de los compiladores

La construcción de los procesadores de lenguajes formales se apoya en una teoría que está vinculada con la Lingüística general. Por ello, es interesante establecer la diferencia entre *lenguajes formales* y *lenguajes naturales*. Es posible distinguir ambos por medio de la simple pregunta: ¿quién surgió primero, el lenguaje o sus reglas gramaticales? [1]. En general, un lenguaje natural es aquel que ha evolucionado con el paso del tiempo con la finalidad de servir a la comunicación humana. Lenguajes como el español, el inglés o el alemán continúan su evolución sin tener en cuenta las reglas gramaticales formales; cualquier regla se desarrolla después de que suceda el hecho de la comunicación, en un intento de explicar, y no determinar, la estructura del lenguaje. Como resultado de esto, pocas veces los lenguajes naturales se ajustan a reglas gramaticales sencillas u obvias.

En contraste con los lenguajes naturales, los formales están definidos por reglas preestablecidas y, por tanto, se ajustan con todo rigor a ellas. Como ejemplo tenemos los lenguajes de programación y los lenguajes matemáticos como el álgebra y la lógica de proposiciones. Gracias a esta adhesión a las reglas, es posible construir programas traductores eficientes para los lenguajes de programación, a la vez que la falta de reglas estrictas dificulta la construcción de traductores para los lenguajes naturales.

Los distintos lenguajes formales que se pueden construir sobre un alfabeto concreto pueden clasificarse de acuerdo con la jerarquía establecida por Noam Chomsky en los años 1950. En el transcurso de sus investigaciones sobre la sintaxis de los lenguajes naturales, Chomsky desarrolla el concepto de gramática libre de contexto. Por otro lado, Backus y Naur desarrollaron una notación formal para describir la sintaxis de algunos lenguajes de programación (notación BNF), que básicamente se sigue utilizando todavía, y que podría considerarse equivalente a las gramáticas libres de contexto.

La teoría de los lenguajes formales tiene una relación inmediata con la teoría de las *máquinas abstractas*. A cada tipo de lenguaje le corresponde una máquina abstracta que lo reconoce. De acuerdo con la jerarquía establecida por Chomsky, los lenguajes formales se pueden clasificar en clases cada vez más amplias que incluyen como subconjunto a las anteriores. La siguiente tabla esquematiza dicha jerarquía:

Lenguaje según Chomksy	Gramática que lo genera	Máquina que lo reconoce
Tipo 0	Tipo 0: sin restricciones o con estructura de frase	Máquina de Turing
Tipo 1	Tipo 1: dependiente del contexto	Autómatas acotados linealmente
Tipo 2	Tipo 2: libre de contexto	Autómatas con pila
Tipo 3	Tipo 3: regular	Autómatas finitos

Como vemos, existe también una asociación directa entre cada lenguaje de la clasificación de Chomsky y la gramática que lo genera.

No debemos olvidar que existen lenguajes que no son de tipo 0 (también llamados *recursivamente enumerables*) y que, por tanto, no son reconocidos por *máquinas de Turing*.

A continuación resumimos las definiciones y principales características de cada uno de los tipos de gramáticas indicados en la tabla anterior:

- **Gramáticas de tipo 0**:

  No tienen restricciones. Son las más generales. Las producciones tienen la forma:

  $$\alpha \rightarrow \beta \ , \ \alpha \in (V_N \cup V_T)^+ \ , \ \beta \in (V_N \cup V_T)^*$$

  Éste es el tipo de gramática cuyo estudio se asocia a la teoría de la *Computabilidad*. Sin embargo, no tiene relevancia en los lenguajes de programación, ya que escribir un analizador sintáctico para una gramática de este tipo es imposible en general, y para los casos en los que es posible, es muy complejo.

- **Gramáticas de tipo 1**:

  La forma de las producciones de estas gramáticas implica que las sustituciones sólo pueden efectuarse en cierto contexto. Tienen la forma:

  $$\alpha A \beta \rightarrow \alpha \gamma \beta \ , \ A \in V_N \ , \alpha\beta \in (V_N \cup V_T)^* \ , \ \gamma \in (V_N \cup V_T)^+$$

  Se introducen algunas limitaciones en la estructura, aunque se permite que el significado de las palabras dependa de su posición en la frase, es decir, de su contexto.

- **Gramáticas de tipo 2**:

  En este tipo de gramática se restringe aún más la libertad de la formación de reglas gramaticales. El significado de una palabra debe ser totalmente independiente de su posición en la frase. La forma de sus reglas es la siguiente:

  $$A \rightarrow \alpha \ , \ A \in V_N \ , \ \alpha \in (V_N \cup V_T)^*$$

- **Gramáticas de tipo 3**:

  Tienen la estructura más sencilla. En la práctica, todos los lenguajes de programación quedan por encima de este nivel, aunque no por ello dejan de tener aplicación en la construcción de compiladores, como veremos. Todas las producciones tienen la forma:

  $$A \rightarrow a \ , \ A \rightarrow aB \quad \text{o } A \rightarrow \lambda$$

  en donde $A, B \in V_N, a \in V_T$.

La mayor parte de los lenguajes de programación pertenecen al tipo 1, aunque gran parte de las reglas de su gramática pueden reducirse al tipo 2, que es por esto el más importante desde el punto de vista del escritor de compiladores, junto con el tipo 3, que define la sintaxis de los símbolos básicos del lenguaje y que, por tanto, se usa en el análisis léxico.

Aunque las gramáticas tipo 1 pueden definir completamente a la mayoría de los lenguajes de programación actuales, su tratamiento es ineficiente. Por esta razón, las partes del lenguaje sensibles al contexto se manejan con la tabla de símbolos y otras herramientas del análisis semántico.

1.5.4. Tipos de compilador

Un *compilador cruzado* (cross-compiler) es un compilador que se ejecuta sobre una máquina (denominada *máquina huésped*) pero genera código para una máquina distinta (llamada *máquina destino*). El programa objeto debe generarse sobre la máquina huésped pero debe descargarse sobre la máquina destino para poder ejecutarlo. Un compilador cruzado es una herramienta necesaria si la máquina destino no tiene memoria suficiente para contener al compilador, o si la máquina destino no está equipada con programas de ayuda al desarrollo[4].

[4]Los compiladores suelen ser programas grandes, que necesitan buenos entornos de desarrollo.

En la figura 1.5 se muestra el diagrama que representa la compilación cruzada de un programa escrito en Java para permitir que se ejecute sobre un microprocesador ARM[5]. Utilizando un compilador cruzado Java-en-ARM se traduce el programa fuente `sort` a un programa objeto equivalente que estará escrito en código máquina ARM. Como el compilador está escrito en código máquina x86, debe ejecutarse sobre una máquina x86. La segunda parte del diagrama muestra el programa objeto que se va a ejecutar sobre una máquina ARM, tras haber sido descargado desde la máquina x86.

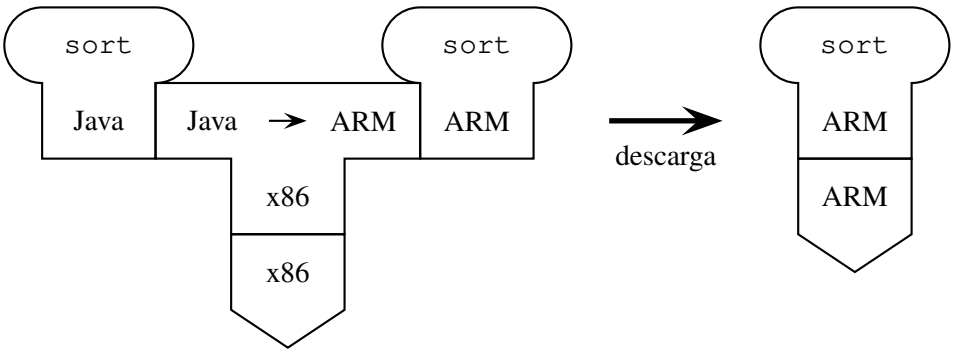


Figura 1.5: Diagrama de un compilador cruzado

Un *traductor de n-etapas* es una composición de *n* traductores que maneja *n-1* lenguajes intermedios. Por ejemplo, dados un traductor Java-en-C y un compilador C-en-x86, se puede realizar una composición de dos etapas, obteniendo un traductor Java-en-x86, como se muestra en la figura 1.6.

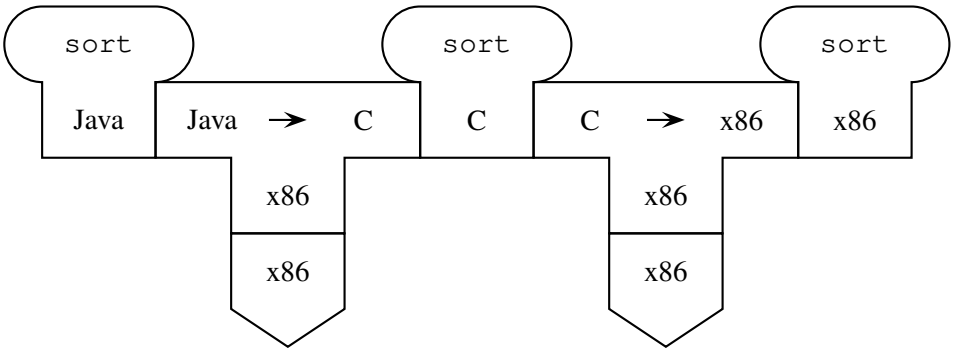


Figura 1.6: Ejemplo de un compilador de dos etapas

Un *compilador de una pasada* genera el código objeto realizando una única lectura del código fuente. Un *compilador de múltiples pasadas* procesa varias veces el código fuente o algunas de las estructuras generadas en las etapas intermedias.

Un *compilador just-in-time* traduce, de forma dinámica, código de una máquina abstracta a código de una máquina real, según se necesite. Suele formar parte de un intérprete (explicado en el apartado 1.6).

Finalmente, un *compilador optimizador* (como se indicó en el apartado 1.5.2.5) modifica el código intermedio o el objeto para mejorar su eficiencia, manteniendo la funcionalidad del programa original.

1.6. Intérpretes

Como hemos visto, un compilador nos permite preparar un programa para que sea ejecutado en una máquina, traduciendo el programa a código máquina. El programa se puede ejecutar a la velocidad que permita

[5] ARM (Advanced RISC machines) es una familia de microprocesadores RISC diseñados por la empresa inglesa ARM Holdings.

el procesador de la máquina. Este método de trabajo no está libre de inconvenientes: todo el programa debe ser traducido antes de que pueda ejecutarse y generar resultados. En un entorno interactivo, la *interpretación* es un método de trabajo más adecuado.

Un **intérprete** es un programa que acepta otro programa (el *programa fuente*) escrito en un determinado lenguaje (el *lenguaje fuente*) y lo ejecuta inmediatamente. Por tanto, un intérprete trabaja cargando, analizando y ejecutando una a una las instrucciones del programa fuente. El programa fuente comienza a ejecutarse y producir resultados desde el momento en que la primera instrucción ha sido analizada. El intérprete no traduce el programa fuente a un código objeto.

La interpretación es un buen método cuando se dan las siguientes circunstancias:

- El programador está trabajando de forma interactiva, y quiere ver el resultado de cada instrucción antes de introducir la siguiente instrucción.
- El programa se va a utilizar una o pocas veces, de modo que la velocidad de ejecución no es importante.
- Se espera que cada instrucción se ejecute una sola vez.
- Las instrucciones tienen un formato simple, y por tanto pueden ser analizadas de forma fácil y eficiente.

En general, la interpretación de programas es lenta. Se estima que la interpretación de un programa fuente escrito en un lenguaje de alto nivel puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina [3]. Por tanto, la interpretación no es interesante cuando:

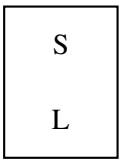
- El programa se va a ejecutar en producción, y por tanto la velocidad es importante.
- Se espera que las instrucciones se ejecuten frecuentemente.
- Las instrucciones tienen formatos complicados, y por tanto su análisis es costoso en tiempo.

Algunos intérpretes conocidos son:

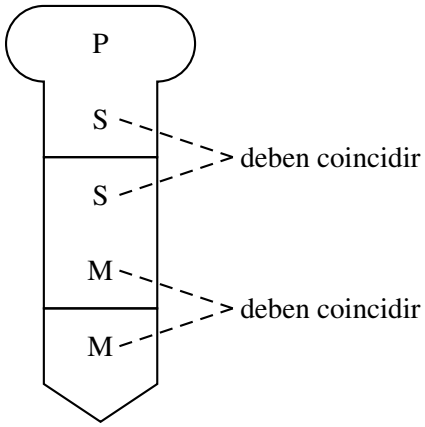
- Intérprete PHP[6]. PHP es un lenguaje que está muy orientado al desarrollo de aplicaciones web. Cuando solicitamos a un servidor web una página PHP, antes de enviar dicha página al navegador, el servidor la procesa a través del interprete de PHP. Éste ejecuta sus instrucciones y es el resultado de esta interpretación lo que termina llegando al navegador.
- Intérprete de comandos Unix (*shell*). Un usuario puede introducir instrucciones para el sistema operativo Unix a través del programa *shell*. Éste lo analiza, extrae un nombre de comando junto con algunos argumentos, y ejecuta el comando por medio de llamadas al sistema. El usuario puede ver el resultado de un comando antes de introducir el siguiente.
- Intérprete SQL[7]. SQL es un lenguaje de consultas a bases de datos. El usuario extrae información de la base de datos introduciendo una consulta SQL, que es analizada y ejecutada inmediatamente. El intérprete SQL encargado de resolver las consultas forma parte del sistema de la base de datos.
- Intérprete Lisp[8]. Se trata de un lenguaje empleado frecuentemente en aplicaciones de Inteligencia Artificial, que asume una misma estructura (árboles) para el código y los datos. Esto permite, por ejemplo, que un programa Lisp pueda generar nuevo código en tiempo de ejecución.

Un intérprete se representa por un rótulo rectangular en cuya base se indica el lenguaje de implementación, mientras que en la parte superior se indica el lenguaje fuente [3] :

[6] Comienza a desarrollarse en el año 1995, llamándose *Personal Home Page tools* (PHP Tools).
[7] Structured Query Language.
[8] El nombre deriva de LISt Processing.



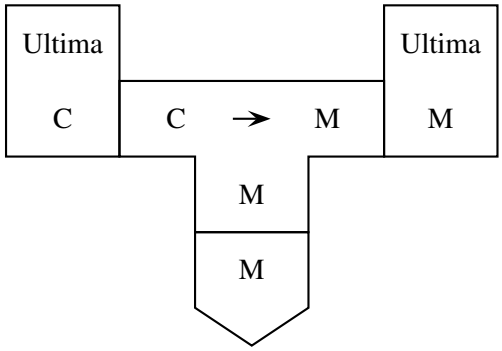
Un intérprete es también un programa, y por tanto puede ejecutarse en una máquina M si y sólo si está escrito en el código máquina de M. Cuando el intérprete se ejecuta, interpreta un programa fuente P que debe haber sido escrito en el lenguaje fuente del intérprete S. Se dice que P se ejecuta *sobre* el intérprete S, como se muestra en la siguiente figura:



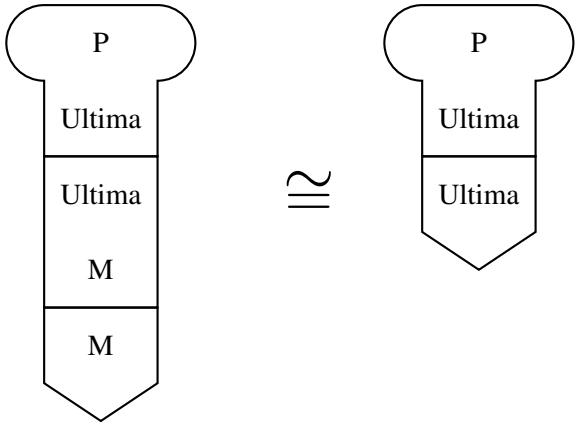
1.6.1. Máquinas reales y abstractas

Los intérpretes indicados en el apartado anterior tienen lenguajes fuente de alto nivel. Sin embargo, también existen intérpretes para lenguajes de bajo nivel. Supongamos que se diseña la arquitectura y el conjunto de instrucciones de una nueva máquina que llamaremos Ultima. Llevar a una implementación real un diseño hardware es costoso y lento. Por tanto, sería deseable no comenzar la construcción hardware hasta que el diseño no se haya verificado completamente.

Existe un método muy simple para verificar un diseño hardware: escribir un intérprete para el código máquina de Ultima en algún lenguaje de alto nivel, como C. Con este intérprete y un compilador de C, se puede traducir el intérprete a un código máquina M, como se muestra a continuación:



Esto nos permite utilizar un intérprete de Ultima en código máquina M, de modo que se pueden ejecutar programas escritos en código máquina de Ultima sobre el intérprete, el cual a su vez se ejecuta sobre una máquina M. En todos los aspectos, salvo en la velocidad, el efecto es el mismo que si el programa se ejecutara sobre Ultima, es decir, las dos configuraciones siguientes son equivalentes:



Este tipo de intérprete se llama *emulador*. No puede utilizarse para medir la velocidad absoluta de la máquina emulada, ya que la interpretación conlleva una ejecución algo lenta. Pero la emulación sí puede permitir verificar el diseño de la máquina emulada.

Como vemos, ejecutar un programa sobre un intérprete es funcionalmente equivalente a ejecutarlo sobre una máquina. El usuario ve el mismo comportamiento en términos de entradas y salidas del programa. Los dos procesos son similares en detalle: un intérprete trabaja mediante el ciclo carga-análisis-ejecución y una máquina trabaja mediante el ciclo carga-decodificación-ejecución. La única diferencia es que el intérprete es un dispositivo software, mientras que una máquina es un dispositivo hardware.

Por tanto, una máquina puede ser vista como un intérprete implementado en hardware. Inversamente, un intérprete puede verse como una máquina implementada en software. A menudo a los intérpretes se les denomina *máquinas abstractas*, para diferenciarlos de su contrapartida en hardware, que son *máquinas reales*. En resumen, una máquina abstracta es funcionalmente equivalente a una máquina real si ambas implementan el mismo código. Igualmente, podemos observar que no existe una diferencia fundamental entre código máquina y lenguajes de bajo nivel. Un código máquina es, precisamente, un lenguaje para el cual existe un intérprete hardware.

1.6.2. Compiladores interpretados

Un compilador puede tardar mucho en traducir un programa fuente a código máquina, pero una vez hecho, el programa generado puede ejecutarse a la velocidad de la máquina. Por otra parte, acabamos de ver que un intérprete permite que el programa comience a ejecutarse inmediatamente, pero su ejecución es lenta.

Un *compilador interpretado* es una combinación de compilador e intérprete que reúne algunas de las ventajas de cada uno de ellos. Este tipo de compiladores se basa en la traducción del lenguaje fuente a un *lenguaje intermedio* diseñado para cumplir los siguientes requisitos:

- tiene un nivel de abstracción intermedio entre el lenguaje fuente y el código máquina.
- sus instrucciones tienen un formato simple, y por tanto pueden ser analizadas fácil y rápidamente.
- la traducción desde el lenguaje fuente al lenguaje intermedio es fácil y rápida.

Por tanto, en una primera etapa se compila rápidamente el programa fuente a un código intermedio, el cual, en una segunda etapa, se puede interpretar a gran velocidad.

El código de la máquina virtual de Java (JVM[9] code) es un lenguaje intermedio del tipo indicado anteriormente. Proporciona potentes instrucciones que corresponden directamente a las operaciones de Java tales como la creación de objetos, llamadas a métodos e indexación de matrices. Por ello, la traducción desde Java a JVM-code es fácil y rápida. Además de ser potentes, las instrucciones del JVM-code tienen

[9] Java Virtual Machine.

un formato tan sencillo como las instrucciones del código máquina, con campos de operación y campos de operandos, y por tanto son muy fáciles de analizar. Por ello, la interpretación del JVM-code es relativamente rápida.

El kit de desarrollo de Java (JDK) consiste en un traductor de Java a JVM-code y un intérprete de JVM-code, los cuales se ejecutan sobre alguna máquina M. Dado un programa P escrito en Java, primero se traduce a JVM-code y a continuación se interpreta el programa objeto JVM-code, como se muestra en la figura 1.7.

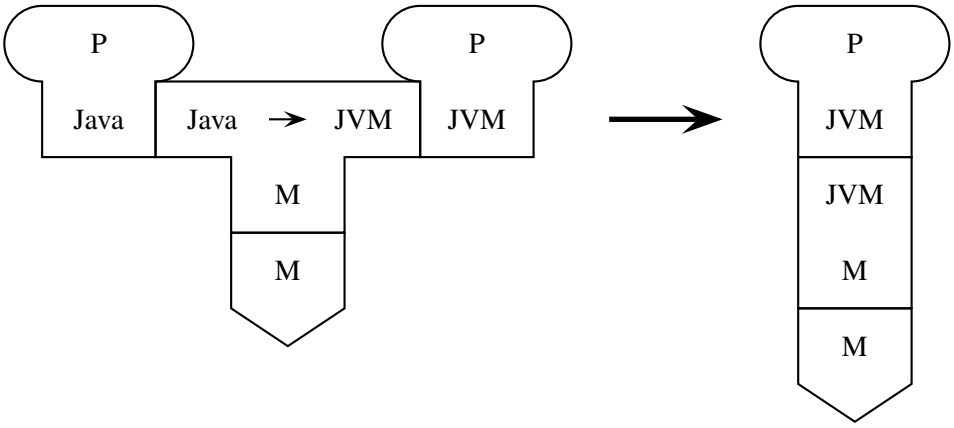


Figura 1.7: Traducción e interpretación de un programa escrito en Java

1.6.3. Compiladores portables

Un programa es *portable* cuando puede ser compilado y ejecutado en cualquier máquina, sin cambios en el código fuente. Se puede medir, de forma aproximada, la portabilidad de un programa como la porción de código que no se tiene que cambiar cuando el programa se mueve entre máquinas diferentes. Lógicamente un programa portable tiene un coste de desarrollo inferior a otro que no es portable.

El lenguaje en el que se escribe el programa tiene una gran influencia en su portabilidad. En un caso extremo, un programa escrito en ensamblador no puede llevarse a una máquina distinta salvo que vuelva a escribirse entero, de forma que su portabilidad es del 0 %. Por otra parte, un programa escrito en un lenguaje de alto nivel es mucho más portable. Idealmente sólo es necesario compilarlo cuando se lleva a una máquina distinta. Podríamos evaluar su portabilidad en el 100 %.

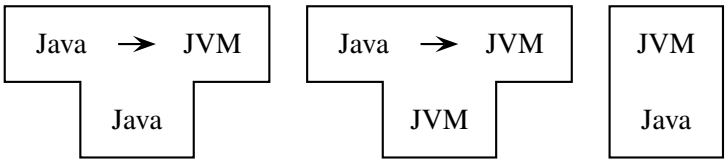
Sin embargo, este ideal es a menudo difícil de alcanzar. Por ejemplo, el comportamiento de un programa puede verse alterado por pasar a una máquina que tenga un conjunto distinto de caracteres o diferente representación aritmética. Escribiendo cuidadosamente los programas en un lenguaje de alto nivel se puede alcanzar una portabilidad del 95-99 %.

Algo similar ocurre con los procesadores de lenguajes, ya que también son programas. Indudablemente, es particularmente importante para un procesador de lenguaje el que sea portable, ya que ello le permite ser ampliamente utilizado. Por esta razón, los procesadores de lenguajes se escriben en lenguajes de alto nivel, tales como Pascal, C o Java.

Sin embargo, un compilador tiene una característica que lo hace no ser 100 % portable. Una de las funciones del compilador es generar código objeto para una máquina concreta, y esta función es dependiente de la máquina por su propia naturaleza. Supongamos que se tiene un compilador de C-en-x86 escrito en un lenguaje de alto nivel. Llevarlo a otra máquina distinta será fácil, pero seguirá generando código máquina para x86. Cambiar el código generado para otra máquina distinta puede requerir la re-escritura de casi la mitad del compilador, de modo que su portabilidad sería del 50 %. Sin embargo, existe una solución parcial

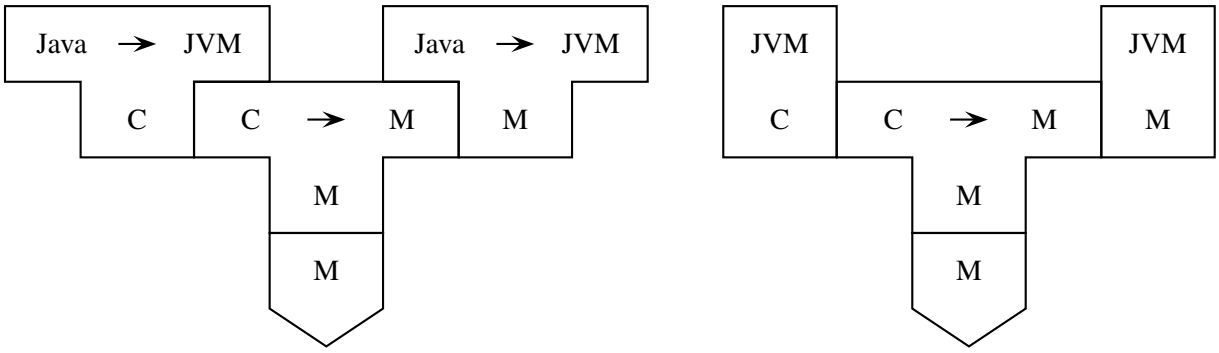
a este problema: el compilador puede generar código intermedio que es potencialmente más portable que el código máquina.

Supongamos que queremos implementar un kit de desarrollo de Java lo más portable posible. Siguiendo la idea anteriormente expuesta, necesitamos un compilador que traduzca desde Java a código intermedio (JVM-code), y evidentemente también necesitamos un intérprete de JVM-code con el que ejecutar los programas compilados. Si además desarrollamos una versión del traductor escrita en JVM-code, también podríamos ejecutar este traductor sobre el intérprete. Por tanto, el kit de desarrollo portable tendría como punto de partida estos componentes:

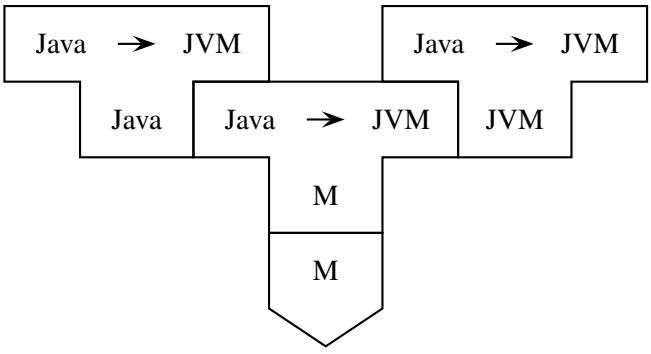


El primer componente indicado se emplea en el proceso de arranque para conseguir el segundo componente. Sin embargo, la configuración anterior no puede conducir a nada práctico por sí sola. No es posible emplear el traductor de Java a JVM-code hasta que no tengamos una versión ejecutable del intérprete sobre alguna máquina. Es necesario encontrar un compromiso entre la portabilidad del kit y su uso práctico.

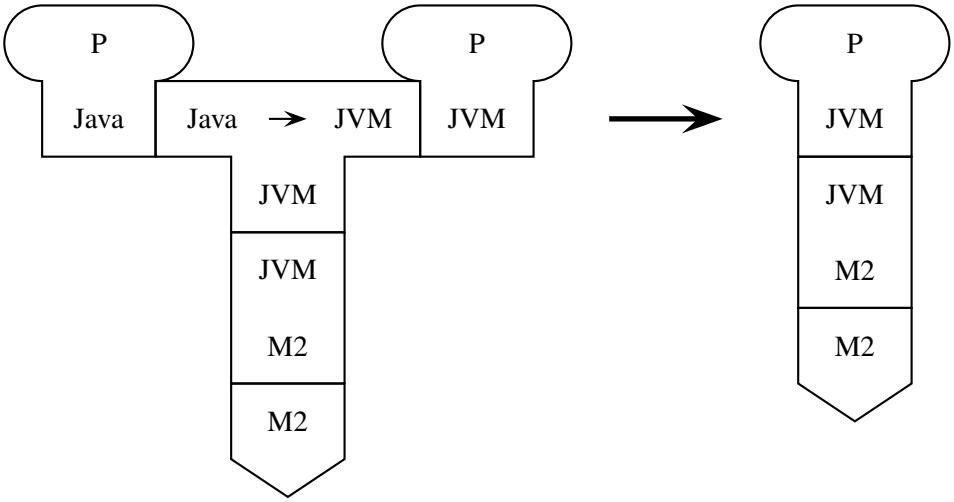
Supongamos que nuestro proyecto para construir el kit portable de Java se está realizando en una máquina M que ya cuenta con un compilador para otro lenguaje de alto nivel como, por ejemplo, C. Lo más sencillo es emplear este lenguaje para hacer una primera versión ejecutable en M del traductor y del intérprete:



El traductor Java-en-JVM-code ejecutable en M se puede emplear para, partiendo de la versión inicial del traductor Java-en-JVM-code implementada en Java, obtener una versión implementada en JVM-code:



En este último paso hemos obtenido una versión del traductor de Java-en-JVM-code que puede ejecutarse sobre un intérprete de JVM-code. Por tanto, si quisiéramos portar el kit de desarrollo a una nueva máquina M2, sólo tendríamos que recompilar el intérprete JVM en la nueva máquina M2. La compilación y ejecución de cualquier programa Java con este kit portable se realizaría de la siguiente forma:



Obsérvese que el traductor Java-en-JVM-code implementado sobre JVM-code no necesita modificaciones para ser usado en la máquina M2. El intérprete JVM-code es mucho menor que el traductor Java-JVM de modo que, incluso en el caso de tener que volver a implementarlo en un lenguaje distinto a C para el que existiese un compilador en la máquina M2, el kit sería portable en torno al 95 %. La re-programación del intérprete es una tarea sencilla para un programador experimentado en el lenguaje de implementación que esté disponible en la máquina M2.