Procesadores de Lenguajes I:

Una introducción a la fase de análisis

J. M. Benedí Ruíz
V. Gisbert Giner
L. Moreno Boronat
E. Vivancos Rubio

Tema 1

Introducción a la compilación

El objetivo de este tema es proporcionar una visión global del proceso de compilación antes de profundizar en sus aspectos concretos, objetivo de los temas siguientes. En este tema se pretende contestar las cuestiones de qué es un compilador, qué hace, cómo se define y cómo se construye. Para ello, en la primera parte del tema se hace hincapié en la íntima relación entre la definición de los lenguajes de programación y el diseño y construcción de sus compiladores asociados. Y en una segunda, se muestra la estructura general de un compilador y su desarrollo en fases.

1.1. Lenguajes de programación y diseño de compiladores

El lenguaje es el medio natural de transmisión de pensamientos e ideas entre los seres humanos. El uso de los computadores implica un proceso de comunicación hombre-máquina, y éste se debe de realizar mediante algún tipo de lenguaje. En informática, un programa es la especificación de una tarea de computación, y un lenguaje de programación es una notación para escribir programas. Los lenguajes de programación sirven como puente para cubrir la enorme separación que existe, en términos de lenguaje, entre las personas con una cierta clase de problemas y los computadores susceptibles de ser utilizados para resolverlos. Esto implica, por un lado, un conocimiento expreso por parte de las personas tanto de la naturaleza del problema como de las particularidades del lenguaje de programación, y por otro, la existencia de un cierto «traductor» que permita expresar un programa escrito en el lenguaje de programación seleccionado, en términos asimilables por la máquina. A esta clase de traductores se les suele llamar **Compiladores**.

La solución computacional a un problema será tanto mas simple y natural de obtener, en tanto en cuanto el lenguaje de programación esté próximo al problema. Esto es, el lenguaje debe contener construcciones que reflejen la terminología y elementos empleados en la descripción del problema, y éstas sean además independientes de la máquina. Un lenguaje de programación

de estas características se denomina comúnmente **lenguaje de alto nivel**, en contraposición con un lenguaje de bajo nivel **lenguaje máquina**.

Desde un punto de vista funcional, un compilador acepta como entrada un programa escrito en un cierto **lenguaje fuente** (usualmente un lenguaje de alto nivel), y genera en un programa escrito en un **lenguaje objeto** (usualmente un lenguaje máquina). A efectos de presentación, este proceso se suele dividir en dos grandes fases (Figura. 1.1): una **fase de análisis** del programa fuente, donde se analiza si cumple con las especificaciones léxicas, sintácticas y semánticas del lenguaje fuente; y una **fase de síntesis** del programa objeto, donde se genera un código equivalente atendiendo a las características de la máquina.

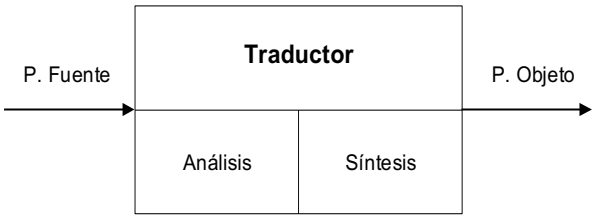

Figura 1.1. Esquema de un traductor (compilador) para un lenguaje de programación.

Como se ha comentado, existe una íntima relación entre la evolución de la potencia descriptiva de los lenguajes de programación y la progresiva complejidad en la especificación y construcción de sus compiladores asociados. Por tanto, la tarea de diseño de un compilador será dependiente de la propia definición del lenguaje de programación correspondiente.

Tal y como muestra la Figura. 1.2, la definición formal de un lenguaje de programación consta de una especificación léxica, sintáctica y semántica:

- La **especificación léxica** define los símbolos válidos del lenguaje que pueden aparecer en el conjunto de programas válidos. El formalismo de especificación léxica utilizado son las **expresiones regulares**.

- La **especificación sintáctica** define el conjunto de programas sintácticamente válidos. El formalismo de especificación sintáctica empleado son las **gramáticas incontextuales**.

- La **especificación semántica** determina tanto las restricciones semánticas del lenguaje como el significado de los programas válidos. En compilación consideraremos que el significado de un programa fuente es el programa objeto equivalente. El formalismo de especificación semántica que utilizaremos serán las **gramáticas de atributos**.

Lenguaje de Programación	⇔	***Traductor***
Especificación léxica *(Expresiones Regulares)*	⇔	Análisis léxico *(Autómatas Finitos)*
Especificación sintáctica *(Gramáticas Incontextuales)*	⇔	Análisis sintáctico *(Autómatas a Pila)*
Especificación semántica *(Gramáticas de Atributos)*	⇔	Análisis semántico *(Esquema de Traducción Dirigido por la Sintaxis)*

Figura 1.2. Relación entre los lenguajes de programación y sus traductores (compiladores) asociados.

Se tiene una idea bastante aproximada de lo que se entiende por una construcción léxica y sintácticamente correcta y además, o quizás por ello, existe un consenso generalizado en el sentido de que la **Teoría de Lenguajes Formales** es un formalismo adecuado para la especificación léxico-sintáctica de los lenguajes de programación.

A diferencia de lo que ocurre con la sintaxis, no existe un claro consenso acerca del formalismo de especificación semántica de los lenguajes de programación. El problema se plantea al intentar definir cuál es el significado de una construcción sintácticamente correcta.

Existen diversos enfoques que abordan la resolución de este problema y que dan lugar, de hecho, a la multiplicidad de modelos semánticos existentes tanto informales como formales: semánticas operacionales, semánticas axiomáticas, semánticas denotacionales, semánticas atribuidas. Esta diversidad de modelos semánticos indica, por un lado, la dificultad del tema, y por otro, que cada modelo trata bien determinados aspectos del problema. En este sentido, y aunque no existe un acuerdo unánime, para las especificaciones semánticas estáticas, se suele emplear un modelo semántico atribuido. La semántica de atributos tiene un enfoque más ligado a la sintaxis y en esencia considera que el significado de cada construcción sintáctica del lenguaje se puede definir en términos de atributos o valores semánticos, y la evaluación de los mismos está guiada por la sintaxis. Este modelo semántico fue introducido en un principio por Knuth, aunque, como siempre, las sucesivas aportaciones han sido innumerables.

El diseño de compiladores no solo consiste en analizar léxica, sintáctica y semánticamente si un programa fuente cumple las especificaciones del lenguaje de programación, sino que también hay que generar código objeto equivalente (Figura1.1). Esta etapa de síntesis, a efectos de eficiencia, se divide normalmente en una generación y optimización de un cierto código intermedio independiente de la máquina y la posterior generación y optimización de código máquina. Un compilador es un producto complejo de ingeniería, y por tanto habrá que considerar también aspectos adicionales como la detección y recuperación de errores, la mejora de los métodos de diseño y construcción de compiladores, el perfeccionamiento de la interfaz con el usuario, etc.

Como se verá a lo largo del curso, el el diseño de compiladores para lenguajes de alto nivel es una tarea compleja. Para darnos cuenta de este hecho, a continuación, nos situaremos en una perspectiva histórica para analizar la evolución de los compiladores y su situación actual.

1.2. Breve historia de los compiladores

1.2.1. Origen y evolución de los compiladores

Aunque en la literatura clásica sobre compiladores existen interesantes y sugestivas revisiones históricas sobre la evolución de los compiladores, nos parece interesante y revelador plantear esta evolución en términos de los problemas fundamentales que se pretendían resolver en cada momento histórico. Así pues, la evolución de los compiladores se podría dividir en tres grandes épocas:

Durante la primera etapa (1945 a 1960 aproximadamente) los lenguajes de programación se desarrollaban lentamente y los computadores eran específicos. El problema fundamental era como generar código eficiente para una máquina concreta. De hecho, el término **compilador** (que se suele atribuir a Grace Murray Hopper en los años 50) como proceso de traducción, se veía como un mecanismo de compilar (reunir) un conjunto de subrutinas de una cierta librería. Sin embargo, lo que entendemos hoy por compilar se denominaba **programación automática**. En ese periodo, había un generalizado escepticismo acerca de las posibles implementaciones de los traductores asociados a los lenguajes de programación (de alto nivel) y a su código generado. Actualmente, la traducción automática de lenguajes de programación es un hecho consumado, pero los traductores de lenguajes de programación se les sigue llamando compiladores.

En esta primera etapa, la mayor parte de los trabajos de compilación estaban relacionada con la traducción de expresiones aritméticas a código máquina. Se suele considerar al traductor del lenguaje FORTRAN el primer compilador que producía un código eficiente. La construcción de este compilador de FORTRAN, supuso una carga de trabajo equivalente a 18 personas-año. La razón por la que se hizo tan largo el proceso era doble: por un lado, el lenguaje se iba diseñando al mismo tiempo que se implementaba, y por otro, el proceso de traducción no era bien conocido ni estaba bien formalizado.

Entre 1960 y 1975 aproximadamente, se produjo un enorme avance en el desarrollo de la Teoría de Lenguajes Formales. La mayor parte de los trabajos estaban relacionados con la especificación y el análisis de la sintaxis de los lenguajes de programación. En ese mismo momento y quizás estimulado por esto, hubo una proliferación de nuevos lenguajes de programación y por tanto una necesidad de construir compiladores para ellos. Esto produjo un deslazamiento en el esfuerzo e importancia que se daba a la fase de análisis en la construcción de los compiladores, convirtiéndose en el problema fundamental de esta época. Todo ello provocó un vigoroso avance en el proceso de sistematización en la construcción de compiladores, y disminuyó sensiblemente el tiempo necesario para construirlos.

La tercera etapa abarca aproximadamente desde 1975 hasta nuestros días. En este periodo, tanto el número de nuevos lenguajes de programación propuestos como el número de tipos de máquinas, de uso normal, diferentes ha disminuido significativamente. Esto ha derivado, de nuevo, en una mayor exigencia en la calidad del código generado. Lo que ha producido un nuevo desplazamiento en el interés hacia la fase de síntesis en general y hacia la generación y optimización de código, en particular.

Igualmente, y prácticamente en el mismo periodo, se desarrollaron nuevos paradigmas de programación: lógica, funcional, orientada a objetos, distribuida, etc. Los requerimientos de ejecución de estos lenguajes son mucho mayores que los correspondientes para lenguajes imperativos. Por lo que, de nuevo, es necesario un mayor esfuerzo en el desarrollo en la generación de código eficiente.

1.2.2. Compiladores hoy

En la actualidad, el diseño de compiladores es una disciplina clásica de la informática que está perfectamente asentada. Hoy en día, un compilador es una herramienta bien conocida y de toda confianza. Sin embargo cabría preguntarse ¿a qué es debido? Probablemente, las razones hay que buscarlas en tres aspectos importantes:

1. **Adecuada estructuración del problema**. La división del compilador en fases (análisis-síntesis) es muy eficiente y permite una correcta separación y resolución de las mismas.

2. **Uso juicioso del formalismo**. Para las etapas de análisis, se han empleado formalismos específicos que han ayudado notablemente al desarrollo de las mismas. Ejemplos típicos son las expresiones regulares, las gramáticas incontextuales y las gramáticas de atributos, que se han usado respectivamente en las fases de análisis léxico, sintáctico y de comprobaciones semánticas.

3. **Uso de herramientas de generación automática de programas**. Ejemplos característicos son los analizadores léxicos, generados automáticamente a partir de las especificaciones léxicas expresadas mediante expresiones regulares; los analizadores sintácticos, generados automáticamente a partir de las especificaciones sintácticas expresadas mediante gramáticas incontextuales; y generadores de código, generados a partir de las descripciones de la máquina. Los programas resultantes son, normalmente, más seguros y fáciles de depurar y, generalmente, mas eficientes que los realizados «a mano».

La importancia de los compiladores hoy en día, y por tanto la necesidad de enseñar/aprender su diseño, está fuera de toda duda. Los compiladores son una herramienta de uso cotidiano para la gestión y programación de los sistemas informáticos y por tanto, un conocimiento profundo acerca de su funcionamiento interno resulta pues fundamental. Además, los conocimientos

adquiridos en el estudio del diseño de compiladores tienen aplicación directa y evidente en otros campos de la informática; citando a Aho y Ullman:

«Aunque es probable que pocas personas realicen o incluso mantengan un compilador para uno de los principales lenguajes de programación, mucha gente puede obtener provecho del uso de un gran número de ideas y técnicas para el diseño general programas»

Por último, la enseñanza de la materia de **Compiladores** nos permite además conseguir que nuestros alumnos conozcan mejor los lenguajes de programación que utilizan, aprendan a programar mejor y se enfrenten a programas de un tamaño medio en los que se aplican métodos formales.

1.3. Descripción de los compiladores

En esta sección se revisan brevemente algunos aspectos relacionados con el proceso de la compilación desde el punto de vista del contexto en el que funciona un compilador, los distintos tipos de compiladores, las herramientas disponibles para la construcción de compiladores y posibles aplicaciones de las técnicas de traducción estudiadas.

1.3.1. Tipos de traductores

Dentro del campo de la traducción de lenguajes de programación, existe una gran variedad de posibles traductores:

- **Compilador**. Traductor que convierte un programa escrito en lenguaje fuente de alto nivel a un programa objeto en lenguaje máquina (Figura 1.3.a).
- **Ensamblador**. Compilador muy sencillo cuyo lenguaje fuente está muy próximo al lenguaje máquina.
- **Intérprete**. Realiza el proceso de la traducción paso a paso, a medida que va ejecutando las instrucciones del programa fuente (Figura 1.3.b).

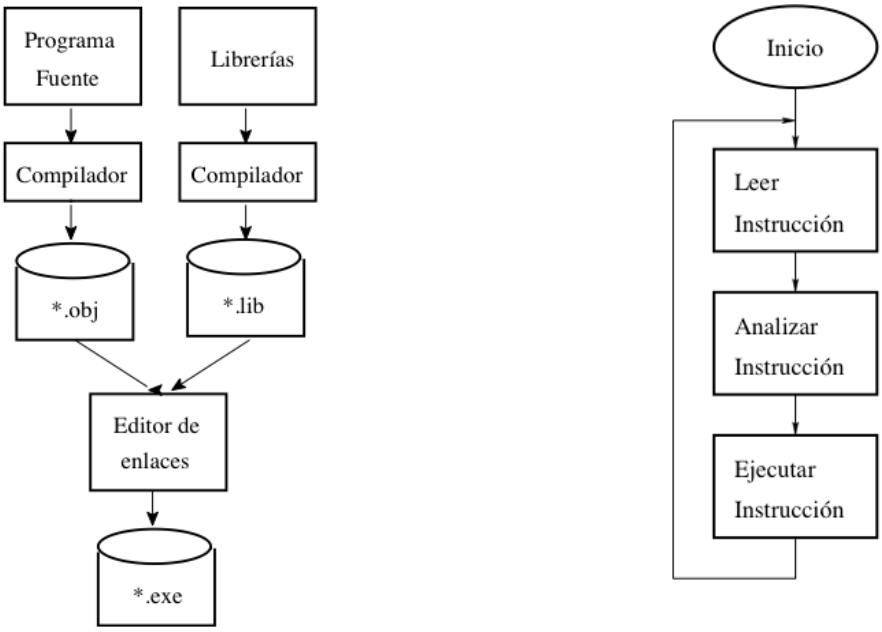

Figura 1.3. Tipos de traductores: (a) Compiladores; y (b) Intérpretes.

- **Intérprete de «bytecodes».** Algunos lenguajes (como «java») combinan compilación e interpretación; tal y como se indica en la Figura 1.4.a. Un programa fuente en «java» es compilado inicialmente en un código intermedio denominado «bytecodes». Posteriormente este código puede ser interpretado por una (diferente) máquina virtual.

- **Compilador «just-in-time».** En relación con el anterior, algunos compiladores (de «java»), para acelerar el proceso traducen el código de «byecode» en el código máquina nativo de la máquina objetivo (Figura 1.4.b).

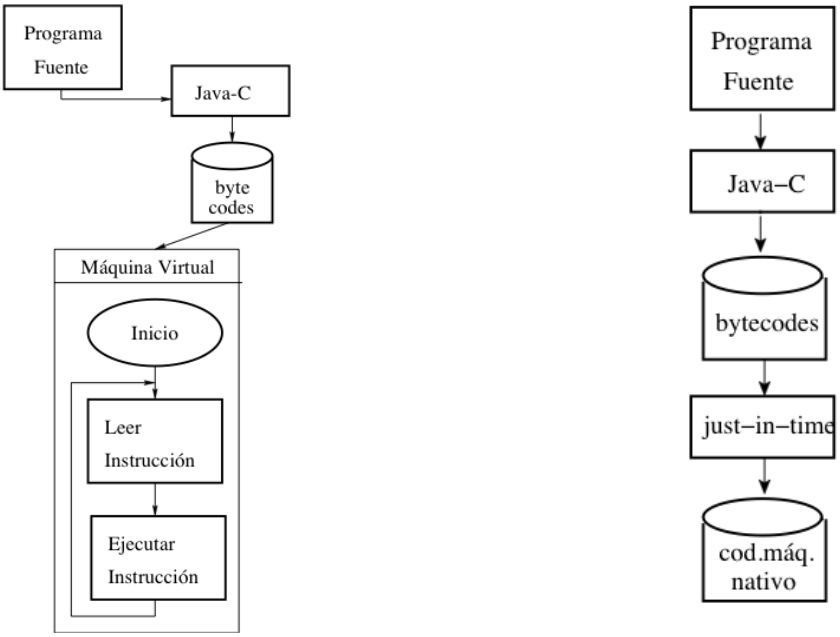

Figura 1.4.- Nuevos tipos de traductores: (a) Intérprete de «bytecodes»; y (b) Compilador «just-in-time.

- **Conversor fuente-fuente**. Traduce un lenguaje fuente de alto nivel a otro (por ejemplo: PASCAL a C). Una aplicación interesante de los traductores fuente-fuente es el desarrollo e implementación de prototipos de nuevos lenguajes de programación.

- **Compilador cruzado**. Genera un código objeto para un computador distinto de aquél en el que se realiza la compilación.

- **Compilador incremental**. Compila un programa; caso de detectar errores, al volver a compilar el programa corregido sólo traduce las modificaciones que se han realizado respecto a la primera versión.

- **Metacompilado**r. Traductor que tiene como entrada la definición de un lenguaje y como salida un compilador para dicho lenguaje.

- **Decompiladores**. Traduce el código máquina a un lenguaje de alto nivel.

1.3.2. Portabilidad y autocompilación

Para ilustrar el problema de la portabilidad definamos primero la **notación T**. Esta notación representa gráficamente los tres lenguajes involucrados en la construcción de un compilador (Figura 1.5): lenguaje fuente, lenguaje objeto y lenguaje en el que está escrito el compilador.

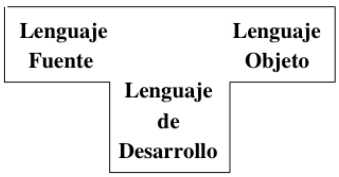

Figura 1.5. Notación T.

Esta notación muestra gráficamente el problema de la portabilidad en la construcción de un compilador. La Figura 1.6.a ilustra dos posibles ejemplos. En el primero de ellos, tenemos un compilador general de PASCAL escrito en C y un compilador de C para una cierta máquina M. Componiendo los compiladores como indica la figura obtenemos el resultado deseado: un compilador de PASCAL para la máquina M. En el segundo (y más realista) ejemplo, tenemos un compilador PASCAL para una máquina M1 escrito en C y un compilador C para una máquina M2. Componiendo los compiladores como indica la figura obtenemos directamente un compilador PASCAL para la máquina M2.

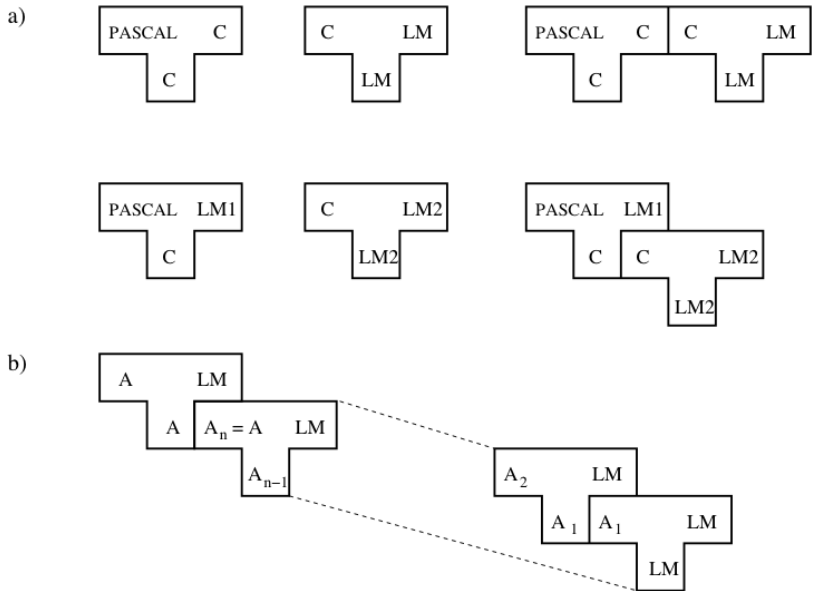

Figura 1.6. Portabilidad y autocompilación.

La autocompilación o técnica «bootstraping», permite construir eficientemente un primer compilador de un nuevo lenguaje A en una máquina M, empleando el mismo lenguaje A para construir el compilador (Figura 1.6.b). El proceso es como sigue: Se define un subconjunto muy reducido de A que llamaremos A(1), y para A(1) se construye su compilador en el lenguaje de la máquina M (o cualquier otro lenguaje que soporte la máquina). Dado el compilador de A(1), se define una ampliación de A(1) llamada A(2), y se construye el compilador de A(2) escrito en A(1). El proceso se repite hasta obtener un compilador de A (A = A(n)) escrito en el lenguaje A. Este proceso que se sigue habitualmente para la construcción del primer compilador de los nuevos lenguajes; por ejemplo el lenguaje C.

1.3.3. Programas relacionados

Existe una serie de programas relacionados con los compiladores, entre los más importantes, podemos destacar:

- **Preprocesadores**. Producen la entrada para un compilador y pueden realizar diversas funciones: procesamiento de macros definidas por el usuario, inclusión de archivos, enriquecimiento de recursos del lenguaje, etc.

- **Ensambladores**. Algunos compiladores producen código ensamblador; por tanto, el resultado de la compilación debe de pasarse a un programa ensamblador para su procesamiento y generación del código máquina.

- **Cargadores y editores de enlaces**. El programa cargador realiza las dos funciones:

- **Edición de enlaces**, para formar un solo programa a partir de diversos archivos de código relocalizable.
- **Carga**, para tomar el código relocalizable generado por el compilador y transformar las direcciones lógicas en direcciones físicas de memoria.

1.3.4. Herramientas para la construcción de compiladores

Las principales herramientas para la construcción de compiladores son:

- **Generadores de analizadores léxicos**. Generan automáticamente un analizador léxico a partir de la especificación léxica expresada en términos de expresiones regulares.
- **Generadores de analizadores sintácticos**. Generan automáticamente un analizador sintáctico a partir de la especificación sintáctica expresada en términos de una gramática incontextual.
- **Otras**. Existen otras herramientas que facilitan la tarea de construir compiladores como son: dispositivos de traducción dirigida por la sintaxis, generadores automáticos de código, dispositivos automáticos para el análisis del flujo de datos, etc.

1.3.5. Otras aplicaciones de las técnicas de traducción

Existen otros campos donde también se aplican las técnicas de traducción que aquí vamos a estudiar, entre los más interesantes cabría citar:

- Formateadores de texto.
- Intérpretes de comandos.
- Intérpretes de consultas a bases de datos.
- Módulos de análisis en aplicaciones que aceptan ficheros de intercambio.
- Lenguaje natural.
- Reconocimiento Sintáctico de formas.

1.4. Estructura de un compilador

El estudio del compilador se suele descomponer en las siguientes fases o módulos (Figura 1.7):

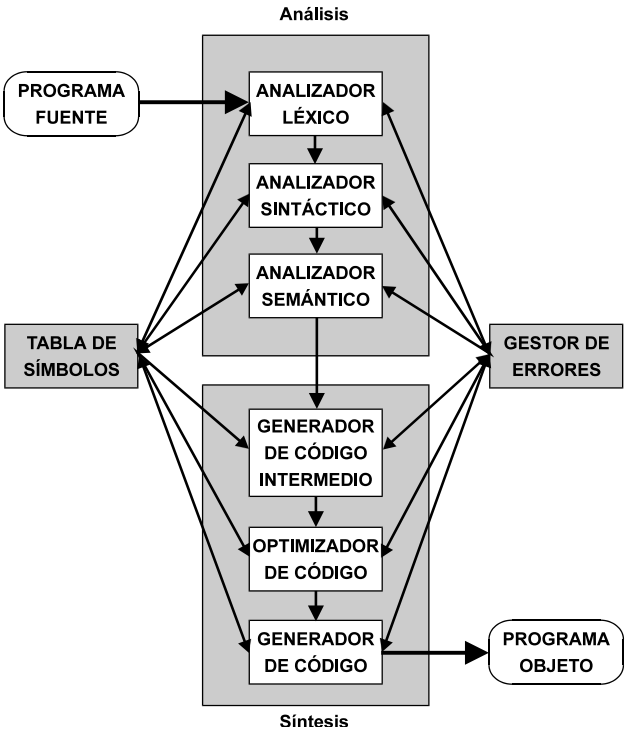

Figura 1.7. Estructura en fases de un compilador.

1.4.1. Análisis léxico

El objetivo de esta fase es leer el programa fuente y generar las unidades léxicas («tokens») encontradas (Figura 1.8). Las principales funciones que se realizan en esta fase de análisis son: detección de los símbolos del lenguaje, realización de las acciones semánticas asociadas al símbolo detectado, generación de los tokens y eliminación de cadenas inútiles.

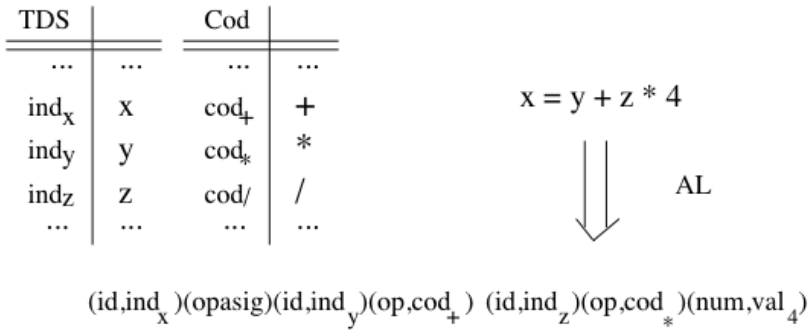

Figura 1.8. Ejemplo del comportamiento del módulo de análisis léxico para un segmento de programa.

1.4.2. Análisis sintáctico

Dada la cadena de tokens generada por el analizador léxico, el analizador sintáctico determina si cumple las restricciones sintácticas expresadas por la gramática del lenguaje; en cuyo caso, proporciona una representación (jerarquizada) del proceso de análisis (Figura 1.9).

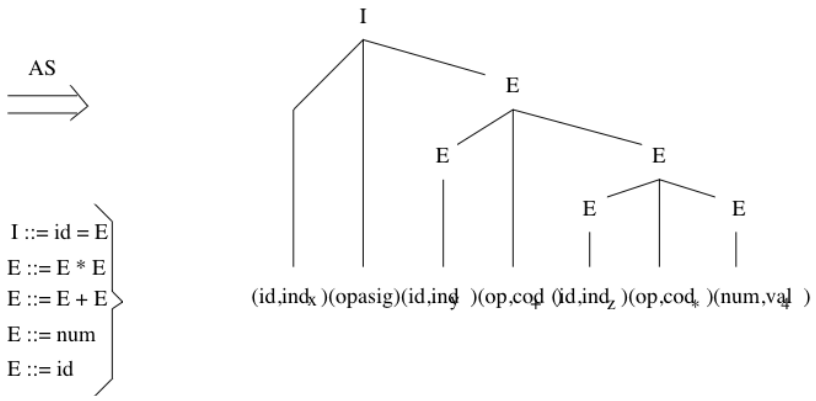

Figura 1.9. Ejemplo del comportamiento del módulo de análisis sintáctico para un **segmento** de programa.

1.4.3. Análisis semántico

El objetivo de este módulo es la verificación de las restricciones semánticas definidas para el lenguaje. Una de las principales componentes del análisis semántico es la comprobación de tipos; es decir la verificación de que cada operación tiene los operandos permitidos por la semántica del lenguaje. En la Figura 1.10 se muestra un posible resultado de la comprobación semántica para el ejemplo de la Figura 1.9 Los nodos del árbol anotado representan las operaciones y los hijos de cada nodo sus posibles operandos. El nodo «entero-a-real» representa la acción semántica de conversión de tipos, considerando que las variables son de tipo real y la constante de tipo entero.

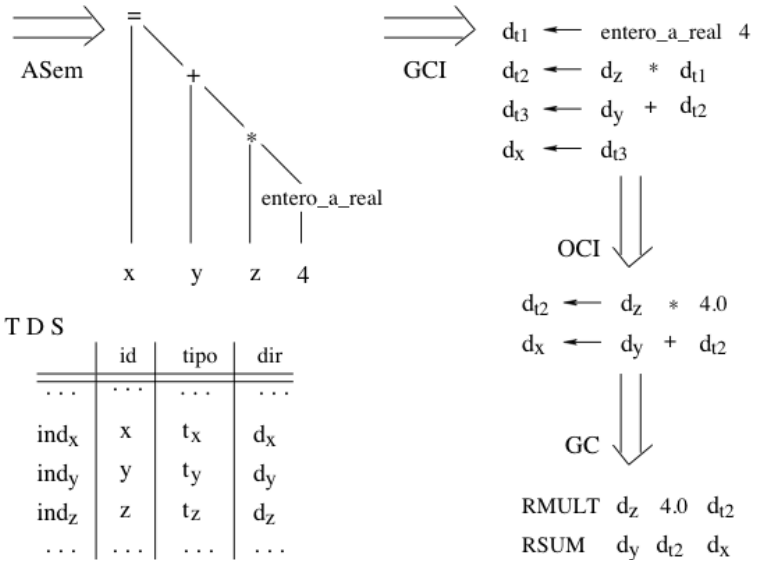

Figura 1.10. Ejemplo del comportamiento de los módulos de análisis semánticos, generación de código intermedio y generación de código máquina.

1.4.4. Generación de código intermedio

El **código intermedio** es un código abstracto independiente de la máquina para la que se generará el código objeto. El código intermedio debe cumplir dos requisitos: ser fácil de producir a partir del análisis sintáctico y ser fácil de traducir al lenguaje objeto. Un ejemplo sencillo, se puede encontrar en la Figura 1.10. Aunque esta fase no es estrictamente necesaria, por razones de portabilidad y de independencia de la máquina es muy recomendable.

1.4.5. Optimización de código (intermedio)

En este módulo se incorporan todas las herramientas disponibles de optimización de código intermedio independiente de la máquina. Se realizan mejoras sobre el código producido similares a las mostradas en el ejemplo de la Figura 1.10.

1.4.6. Generación y optimización de código

En esta última etapa se genera código máquina intentando aprovechar al máximo las prestaciones de la arquitectura de la máquina para obtener un código lo más optimizado posible. En la Figura 1.10 se ilustra un ejemplo sencillo para un supuesto lenguaje máquina.

1.4.7. Tratamiento de errores y gestión de la tabla de símbolos

En la tabla de símbolos se almacena toda la información (atributos) disponible para cada uno de los objetos definidos en el programa fuente analizado (Figura 1.10).

La detección de errores se hace en cada uno de los módulos del compilador. El tratamiento de errores debe de informar sobre los errores detectados en el programa fuente y continuar con el proceso de análisis en busca de otros errores.