Consideraciones generales

Los compiladores son citados y descriptos en numerosas oportunidades a lo largo de este libro. En el Capítulo 1, son presentados como una de las más importantes aplicaciones de las máquinas abstractas (*Construcción de compiladores*). En el Capítulo 2, se hace referencia a la contribución de Noam Chomsky al desarrollo de lenguajes de programación de alto nivel (*Gramáticas formales*); en el Capítulo 4, se presentan los analizadores léxicos (*Gramáticas regulares y autómatas finitos*); en el Capítulo 5, se presentan los analizadores sintácticos (*Autómatas con pila asociados a una gramática*) y, finalmente, el Capítulo 8 está destinado a una introducción a la Semántica de Lenguajes. A pesar de ello, se considera aún necesario incluir una presentación conjunta de los conceptos generales referidos a los compiladores y ese es el motivo de este Apéndice A.

Lo expresado evidencia la importancia que se le reconoce a los compiladores y esto merece ser comentado. El proceso de desarrollo de software incluye numerosas etapas, desde la especificación de los requerimientos del sistema hasta su puesta en servicio y el posterior mantenimiento, pasando por las etapas de análisis, diseño, programación, verificación, integración y validación, por citar las más importantes. Son todas etapas fundamentales en la construcción de buen software y en todas ellas se utilizan, cada vez en mayor medida, herramientas específicas.

Sin embargo, en este proceso, hay una etapa que es central, ya que en ella se construye el producto software (*programación*). Para ello, se utiliza una herramienta, considerada principal, que es la que hace que esta tarea sea posible (*el compilador*). En efecto, el compilador permite utilizar lenguajes de alto poder expresivo, incluyendo el lenguaje visual, y su misión es convertir estos programas a un lenguaje compatible con el hardware, que es el encargado de ejecutarlos.

En resumen, el compilador es una herramienta de desarrollo de software destinada a traducir un programa escrito en un lenguaje de alto nivel (*programa fuente*) en otro programa escrito en lenguaje máquina (*programa objeto*). Nótese que, por el diferente poder expresivo de ambos lenguajes, se enfatiza que se trata de una traducción o conversión que requiere una valoración semántica y no de una mera traducción literal.

Resulta difícil aceptar que no hace mucho, apenas unas décadas atrás, los computadores debían ser programados con códigos binarios que representaban las instrucciones individuales reconocidas por la unidad central de proceso. Estos eran los denominados "*Lenguajes de Primera Generación*" y su utilización era difícil, problemática y muy susceptible a errores. Además, el mantenimiento de los programas escritos en tales lenguajes era una tarea tremendamente ardua y costosa.

Posteriormente, el código de máquina fue reemplazado por nombres simbólicos o "abreviaturas nemónicas", dando lugar así a los "*Lenguajes de Segunda Generación*" o "*Lenguajes ensambladores*". Se comenzó a observar una incipiente tendencia hacia la estructuración de los programas y el uso de estas abreviaturas nemónicas facilitaban el uso de las instrucciones y el mantenimiento de los programas. Sin embargo, se mantenían las dificultades propias de lenguajes tan elementales, como son la fuerte dependencia de la máquina y la necesidad de muy profundos conocimientos de su arquitectura. En síntesis, la programación con tales lenguajes continuaba siendo una tarea artesanal reservada para un selecto grupo de especialistas.

Para sustituir a los lenguajes ensambladores se crearon los "*Lenguajes de Tercera Generación*" o "*Lenguajes de alto nivel*". Estos lenguajes permitieron alcanzar un mayor nivel de abstracción, apropiado para la especificación de datos, funciones y su control, todo ello en forma independiente de la máquina. También hicieron posible una rápida evolución de la programación con la incorporación de técnicas orientadas a la obtención de menores tiempos de desarrollo y mayor claridad en el código. Así, progresivamente, se impusieron nuevos conceptos, tales como los de programación estructurada, diseño modular, programación orientada a objetos, programación visual, etcétera.

Al hacerse una referencia cronológica a estos Lenguajes de *alto nivel* debe comenzarse por el Fortran (***For****mula* ***Tran****slation*) que fue presentado a mediados de los años cincuenta y luego estandarizado en sucesivas oportunidades (*Fortran IV*, *Fortran 66*, *Fortran 77* y *Fortran 90*). La

primera versión del Fortran tuvo la virtud de demostrar que un compilador de lenguaje de alto nivel era capaz de generar código eficiente apto para ser reconocido por una máquina. Ocurre que hasta ese momento había una controversia sobre la verdadera posibilidad de generar buen código por parte de un compilador. Debido a la popularidad que alcanzó este lenguaje en el mundo científico, y las vastas librerías existentes para todo tipo de aplicación técnica, el Fortran no solo es el lenguaje más antiguo, sino que mantiene plena vigencia y la tiene asegurada por muchos años más. El otro lenguaje casi tan antiguo como el Fortran y todavía vigente es el Cobol (***Co**mmon **B**usiness **O**riented **L**anguage*). Orientado a aplicaciones administrativas fue estandarizado en 1968, 1974 y 1985, convirtiéndose en la piedra angular de los desarrollos de sistemas de gestión hasta prácticamente nuestros días. Como contrapartida, tradicionalmente ha presentado una pobre capacidad de transportabilidad debido a sus diferentes implementaciones, que fueron consecuencia de variadas interpretaciones de estos estándares.

Una vez mencionados el Fortran y el Cobol, sigue una extensa lista de lenguajes de alto nivel, que incluye entre muchos otros, el Algol, Basic, APL, PL/1, Pascal, C, ADA, Modula 2, y las recientes implementaciones visuales de la mayoría de ellos. Un comentario especial merece el Algol, en el que cabe destacar su enfoque innovador con la introducción de las estructuras de bloques, declaración explícita de tipos de variables, recursividad y asignación dinámica de memoria, todo ello hace ya más de cincuenta años. Además, como ya fue anticipado en el Capítulo 1, en el *Algol60* John Backus y Peter Naur usaron por vez primera las reglas de reescritura o reglas BNF, tanto en el desarrollo del propio compilador como en su formalización.

Indudablemente, la evolución hacia lenguajes más poderosos viene estando asociada a la disponibilidad de sus correspondientes compiladores, y de allí la gran importancia que estas herramientas han adquirido en la corta historia de la informática.

Conceptos relacionados

Habiendo definido al compilador como una herramienta de conversión de programas *fuentes*, escritos en lenguajes de alto nivel, a programas *objeto* expresados en lenguaje máquina, es ahora necesario conocer una serie de conceptos relacionados.

Editor de programas: Los programas fuentes son archivos de texto, no estructurados, que contienen el código expresado en un lenguaje superior. Esta información debe ser cargada por el usuario, almacenada, ordenada y presentada visualmente al programador a partir de un estructuramiento lógico que se apoya en caracteres de control, esencialmente <CR>, <LF>, <TAB> y otros comandos auxiliares, que tabulan y separan las sentencias del programa en diferentes renglones. Algunos lenguajes disponen de *Editores de Programas* específicos que reconocen variables, constantes, instrucciones del lenguaje, palabras reservadas, directivas al compilador y diagnósticos de errores, utilizando diferente simbología que incluye variados tipos de letras y colores.

Compilación*:* Se refiere al proceso de convertir un programa fuente en un programa objeto, para lo cual el compilador comienza por verificar la integridad y ausencia de errores del primero.

Compilación en varias pasadas*:* Define la cantidad de veces que un compilador debe leer el programa fuente previo a la generación del programa objeto.

Intérprete*:* Se denomina así a una herramienta o módulo que interpreta y ejecuta las sentencias de un programa fuente una tras otra, sin generar un programa objeto.

Preprocesador*:* Es un módulo que tiene la finalidad de modificar o completar al programa fuente previo a ser leído por el compilador. En algunos casos, esto se realiza insertando el contenido de otros archivos y en otras expandiendo el código a partir de macroinstrucciones o directivas ajenas al lenguaje compilado. Conceptualmente, el preprocesador es un módulo independiente del compilador, aunque es frecuente que estén implementados en forma conjunta.

Conversor Fuente-Fuente*:* Tienen por finalidad la conversión de un programa fuente desde un lenguaje de alto nivel a otro. Estos conversores tienen amplia difusión cuando aparecen en el mercado nuevos lenguajes, pudiendo citarse como ejemplos los conversores Pascal ⇨ C y los conversores C ⇨ Java. Su finalidad es facilitar a los nuevos usuarios la migración de un lenguaje a otro.

***Ensamblador**:* Se denomina así a un compilador que tiene por lenguaje fuente a uno de 2ª generación, por lo que se trata de un procesador sencillo que guarda una relación 1 a 1 entre entrada y salida. La capacidad de estos compiladores es, en algunos casos, ampliada con macroinstrucciones que facilitan la definición del programa fuente.

Administrador de librerías: Es una herramienta que permite gestionar en una librería las partes de un sistema que han sido compiladas por separado y que están destinadas a integrar una aplicación.

Depurador: Llamado *debugger* en su idioma original, es un módulo usado para facilitar las pruebas y eliminar errores de los programas.

Enlazador: Llamado *task builder* o *linkeditor* en su idioma de origen, tiene la misión de construir el programa ejecutable a partir del programa objeto, las librerías que pueda tener la aplicación y la librería del sistema operativo. Esta última suele ser reconocida como *run time library*. El programa ejecutable es almacenado en un archivo cuya estructura y denominación (normalmente extensión EXE) son acordes a las exigencias específicas de cada sistema operativo.

Librería de enlace dinámico: Se trata de librerías que, en lugar de ser incorporadas al programa ejecutable por el enlazador, son incorporadas a la aplicación en tiempo de ejecución cuando son necesarias. Podrían ser consideradas una extensión del programa ejecutable y son llamadas *Dynamic Link Libraries* en su idioma de origen. La extensión de estos archivos es DLL y normalmente contienen la mayor parte de las facilidades que ofrecen los sistemas operativos. Permiten que componentes básicos sean compartidos por numerosas aplicaciones y reducen el tamaño de los archivos de los sistemas.

Tipos de compiladores

Sobre la definición básica del compilador ya enunciada se presentan diversas variantes, que dependen tanto de la naturaleza del lenguaje fuente como del tipo de código generado, las que son definidas a continuación.

Compilador cruzado: Es el compilador que genera programas objeto que están destinados a ser ejecutados en computadores diferentes de aquél en el que se lo ha compilado. Ejemplos de compiladores cruzados son aquellos que operan en computadores personales (PC) y generan programas objeto que están destinados a diversos microcontroladores de aplicaciones especiales, tales como balanzas, módems, teclados, etc. En el caso de los teclados de las PC, estos utilizan pequeños micros (Intel 8048 o equivalentes) cuyo programa objeto es generado con un compilador cruzado. Para poder ejecutar uno de estos programas en el computador donde fue compilado, es necesario un emulador, tal como ya fue visto en el Capítulo 7.

***Autocompilador**:* Es un compilador en que su propio programa fuente está escrito en el mismo lenguaje que el de los programas fuentes que admite. Un ejemplo son los compiladores de lenguaje C más recientes, que normalmente han sido escritos en el mismo lenguaje.

***Metacompilador**:* Se trata de un compilador capaz de admitir programas fuentes escritos en diversos lenguajes. Un ejemplo típico son los *MetaAssembler*, que son compiladores preparados para reconocer instrucciones de lenguajes ensambladores de diferentes procesadores y generar en consecuencia los correspondientes programas objeto. En estos casos, se debe identificar la variante del lenguaje fuente a ser utilizada.

***Decompilador**:* Se trata de un módulo destinado a operar en el sentido inverso al de un compilador, es decir, convertir programas escritos en lenguaje máquina a programas fuentes en lenguajes de alto nivel. Los desensambladores son los decompiladores más simples y usuales, que convierten lenguaje máquina en lenguaje ensamblador.

Compilador optimizador: Al generar el programa objeto se tiene en cuenta la necesidad de mejorar el rendimiento del sistema, manteniendo la funcionalidad original. Si bien puede haber distintos criterios de optimización, los objetivos principales son la reducción del tamaño del programa ejecutable, la reducción de la demanda de memoria y la rapidez de operación, los que normalmente son excluyentes entre sí. La mayoría de los compiladores modernos ya tienen incorporadas diversas opciones de optimización.

***Compilador gráfico**:* Son aquellos que admiten programas objeto representados en forma simbólica; normalmente como esquemas lógicos o eléctricos. Pueden citarse como ejemplo los *Controladores Lógicos Programables* (PLC) usados en la industria, cuyos programas objeto pueden ser expresados en lenguaje simbólico/gráfico. Se trata además de compiladores cruzados, ya que el programa fuente es compilado en una PC y luego transferido a la unidad de control (PLC) mediante una puerta de comunicaciones.

Compilador intérprete: Se trata de compiladores que generan los programas objeto en un lenguaje intermedio, que luego son interpretados en el momento de la ejecución. Con esto, se obtiene total transportabilidad de los programas objeto ya que las particularidades de las plataformas son incorporadas en la etapa de interpretación. El mejor ejemplo es el compilador del lenguaje Java, que convierte los programas fuentes generando un código intermedio denominado *bytecode*, que luego es, a su vez, interpretado por una Máquina Virtual de Java (JVM).

Ambiente integrado de desarrollo* (IDE)*: Son sistemas interactivos que incorporan al compilador servicios complementarios, tales como un editor de programas fuentes, facilidades para interpretar los programas y ejecutarlos *paso a paso* o en forma parcial, identificar errores (depuración o *debugging*) y gestionar las librerías de los proyectos.

Los ambientes integrados de desarrollo fueron factibles a partir de la disponibilidad de suficiente capacidad de memoria y de la posibilidad del usuario de operar en forma interactiva a través de las pantallas de video, mouse y teclado. De esta manera, se convirtieron en herramientas que contribuyeron enormemente a facilitar y hacer más eficiente el desarrollo de software, al extremo que todos los lenguajes modernos disponen de su IDE.

Compiladores e intérpretes

Los intérpretes tuvieron mucha vigencia en los primeros tiempos de la computación y han vuelto a recuperarla recientemente. Una de las causas de este predominio inicial de los intérpretes fue la reducida memoria de los computadores. La ventaja que ofrecían los intérpretes era la posibilidad de leer y ejecutar los programas en forma progresiva, sin necesidad de cargar en la memoria al programa fuente completo ni la de generar luego en memoria un programa objeto. Con los compiladores, por el contrario, debían convivir en la memoria el propio compilador con ambos programas y esto se convertía en una restricción importante en el caso de grandes aplicaciones.

Para entender el problema, es necesario tener presente las severas limitaciones de memoria con las que habitualmente se trabajaba. Un ejemplo, como tantos otros, describe al problema en toda su magnitud: a mediados de los años 1970 los sistemas de cálculo estructural, que incluían varios miles de instrucciones Fortran, Algol o PL/1, debían ser compilados y ejecutados en computadores de tan solo 32 Kbytes de memoria RAM. No es un error de impresión, aunque cueste creerlo la memoria disponible era 1/32 de un megabyte. Seguramente, mucho menos que la memoria disponible en el teléfono celular más elemental de la actualidad.

Superadas las limitaciones de memoria con el avance de la tecnología, los compiladores impusieron su ventaja al revisar y tratar al programa fuente en forma integral; generando diagnósticos de errores completos que agilizaban el desarrollo de programas y evitaban que se detectaran errores de sintaxis o semántica al momento de la ejecución (clásico de las aplicaciones desarrolladas en intérpretes Basic, Fox, DBase y otros lenguajes habituales de los años ochenta). Debe, sin embargo, reconocerse lo atractivo que resultan los intérpretes en el testeo de nuevos algoritmos o en el desarrollo de pequeñas aplicaciones. La posibilidad de ejecutar progresivamente el programa, aun cuando una parte de él puede todavía no haber sido escrita o estar incompleta, ofrecía una clara ventaja frente al compilador que requiere una presentación completa y rigurosa del código fuente.

Mucho más recientemente han retomado vigencia los intérpretes y esto es debido a dos razones independientes: *i*) se consolidaron las plataformas de desarrollo híbridas, que incluyen etapas de compilación e interpretación, tal como es el caso de Java ya referido en el punto anterior. Aquí la transportabilidad del software es uno de los principales justificativos para adoptar esta solución. *ii*) las arquitecturas clientes-servidor mostraron rápidamente las ventajas de transmitir código intermedio a ser interpretado directamente en el equipo *cliente*, concepto en el que se apoyan las aplicaciones Web. En efecto, la mayoría de los servicios de Internet son tipo de

cliente-servidor, donde la acción de visitar un sitio Web implica que el servidor transfiere al navegador del cliente la información para ser interpretada y desplegada localmente.

Otro aspecto que debe ser considerado es la rapidez de ejecución y ésta es la principal desventaja de los intérpretes. Por este motivo, los intérpretes nunca son utilizados en aplicaciones que incluyan el procesamiento matemático de grandes volúmenes de datos ni tampoco en aplicaciones de tiempo real.

En resumen, comenzaron teniendo primacía los compiladores, luego resultaron aconsejables los intérpretes, se regresó a los compiladores y en la actualidad ocupan un lugar destacados los lenguajes interpretados, los que alguna vez fueron considerados obsoletos. Como puede comprobarse, la evolución de la tecnología hace que en informática no sea posible hacer pronósticos definitivos.

Notación T

Tal como fue anticipado, todo compilador involucra tres lenguajes de programación: *i*) el del programa fuente "*F"*, que se desea compilar, *ii*) el del programa Objeto "*O"*, que se desea generar y *iii*) el lenguaje de implantación "*I"*, que es aquél en el que el propio compilador fue escrito. Esto resulta convenientemente representado con un esquema denominado *Notación T*, propuesta por H. Bratman, que permite representar el vínculo entre los tres lenguajes, tal como se muestra en la Figura A.1:

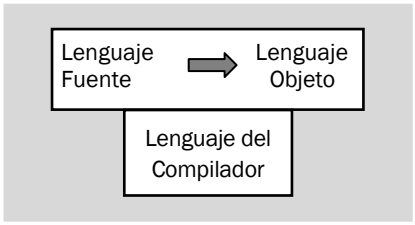

Figura A.1: Representación de un compilador en notación T.

Debe recordarse que el compilador es, a su vez, un programa y como tal debió ser compilado, para lo cual éste requirió alguna vez de un compilador. Supóngase como ejemplo un compilador Java destinado a generar programas para PC y escrito en Lenguaje C. Para desarrollar ese compilador Java, fue necesario un compilador C, que se asume fue uno de los primeros compiladores C y; por lo tanto, escrito en Modula2. Para ello, a su vez, fue necesario disponer de un compilador Modula2, y así continúa una cadena que se inicia con un compilador primitivo que habrá sido laboriosamente escrito en código máquina. Esta sucesión de compiladores puede ejemplificarse en el esquema de la Figura A.2:

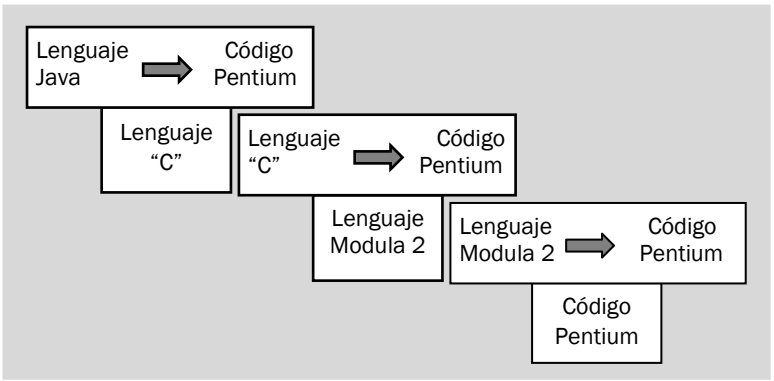

Figura A.2: Encadenamiento de compiladores en notación T.

Se deduce entonces que la hipotética aparición de una nueva máquina basada en un microcomputador incompatible con todo lo existente requeriría armar una cadena de este tipo, a partir del código propio de la máquina y hasta alcanzar los lenguajes de alto nivel deseados. Indudablemente que este es un esfuerzo progresivo, ya que para las etapas iniciales se requiere la intervención de versiones muy simplificadas de compiladores de lenguajes de nivel intermedio.

Contexto del compilador

El contexto del compilador incluye los diversos recursos necesarios para hacer posible la generación de programas ejecutables a partir de los correspondientes programas fuentes.

Estos recursos, que toman la forma de módulos o herramientas de desarrollo, ya han sido descriptos anteriormente y se presentan en forma esquemática en la Figura A.3. Se recomienda al lector reconocer estos módulos, la función que cada uno cumple en el proceso de generación de programas y la interdependencia con los demás componentes del contexto.

Aquí cabe acotar que la totalidad de los componentes que están incluidos en el contexto de un compilador forman parte de los ya definidos Ambientes Integrados de Desarrollo (IDE). Es decir, los IDE son herramientas que permiten editar, compilar, corregir y ejecutar un programa. Para esto último, se dispone de facilidades de ejecución "paso a paso", detención en cierto punto (*break*), evaluación de las variables, reasignación de valores a las mismas y otras opciones.

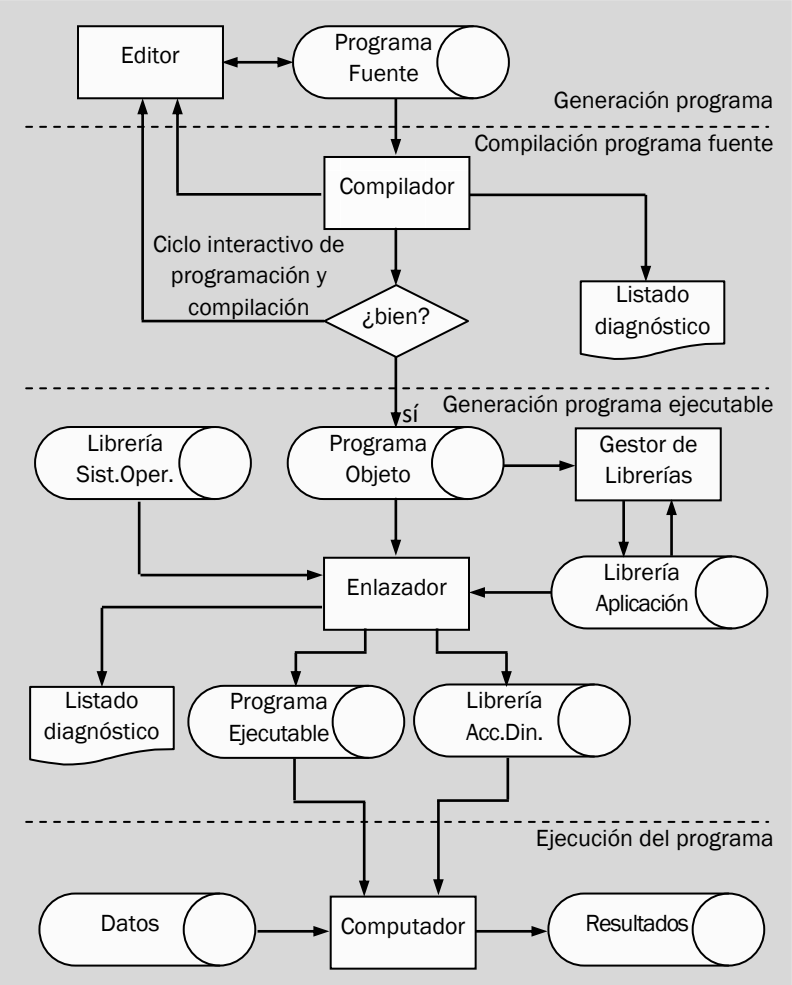

Figura A.3: Contexto de un compilador.

Estructura y componentes del compilador

En la compilación, se distinguen dos etapas principales, que son las de análisis y de síntesis mostradas en la Figura A.4. En la etapa de análisis, se divide el programa fuente en sus elementos componentes y se crea una representación intermedia del mismo. En la etapa de síntesis, se reconstruye el programa objeto a partir de esta representación intermedia.

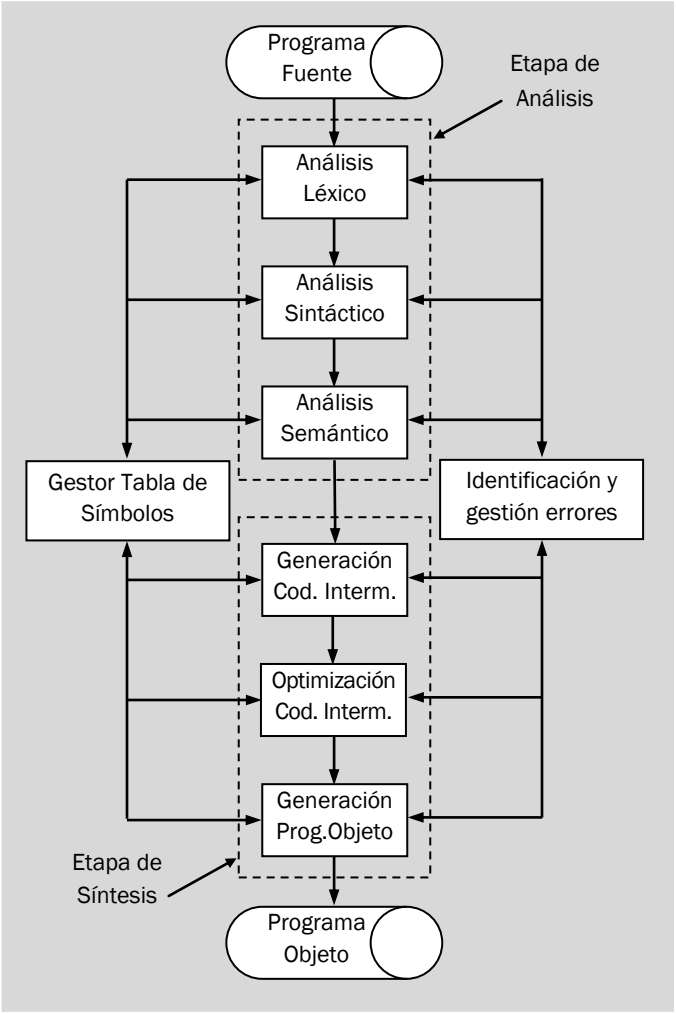

Figura A.4: Estructura conceptual de un compilador.

Tal como se muestra en el esquema anterior, las etapas de análisis y síntesis pueden ser divididas en tres fases cada una. Es decir que el proceso de compilación reconoce el cumplimiento de seis fases claramente distinguibles desde un punto de vista conceptual. Nótese que la etapa de análisis es claramente dependiente del lenguaje del programa fuente y la etapa de síntesis lo es del tipo de procesador al que está destinado el programa objeto. Por este motivo, con solo combinar distintas etapas de análisis con una misma etapa de síntesis se pueden construir compiladores para varios lenguajes fuentes destinados a un mismo procesador. En sentido inverso, combinando una etapa de análisis con diferentes etapas de síntesis se dispondrá de compiladores de un mismo lenguaje fuente destinados a diferentes procesadores. Debe observarse que para que esto sea posible la interfaz entre las etapas de análisis y síntesis debe estar normalizada.

Entrando más en detalle, en la etapa de análisis se distinguen las fases de *análisis léxico*, *análisis sintáctico* y *análisis semántico*. En la etapa de síntesis, las fases son las de *generación de código intermedio*, *optimización* y *generación del programa Objeto*. A lo largo de estas fases, dos módulos esenciales están permanentemente activos en el compilador, que son el destinado a gestionar la Tabla de símbolos y el Administrador de Errores.

Aquí cabe aclarar que, si bien los módulos aparecen encadenados secuencialmente, sus fronteras son más bien conceptuales y no son tan claras en la implementación. Además, el proceso es altamente complejo e implica avances y retrocesos entre módulos contiguos. Es necesario destacar que a la complejidad inherente del problema se le suma la necesidad de hacer mínima la demanda de recursos, tanto en tiempo de proceso como en memoria ocupada.

Análisis léxico

También se denomina análisis lineal o de exploración y el módulo destinado a esta tarea es denominado analizador lexicográfico o *scanner*. En esta fase, la cadena de caracteres que constituye el programa fuente es leída carácter a carácter, para identificar y agrupar sus componentes léxicos. Estos componentes léxicos son secuencias de caracteres (cadenas) que tienen un significado colectivo (variables, constantes, instrucciones del lenguaje, operadores aritméticos, etc.) y son denominados ***tokens***. Para identificar unívocamente un tipo de ***token***, se reconoce a la expresión regular como patrón léxico. Ésta es una expresión abstracta que permite referenciar al conjunto de todos los *tokens* que se ajustan a él. Luego, una cadena de caracteres específica que se ajuste al patrón léxico de un cierto tipo de *token* es denominada lexema. Algunos ejemplos se presentan en la Tabla A.1.

Componentes léxicos o *tokens*	Lexemas	Descripción
Identificadores	ii, total0, valor3, ...	Primer símbolo alfabético seguido de otros símbolos o dígitos
Operadores relacionales	<, <=, ==, >, >=, !=	Símbolos menor, igual o mayor y sus combinaciones
Operadores aritméticos	+, -, *, /, %	Símbolos de operaciones aritméticas básicas
Operadores de manipulación de bits	&, |, ^, ~, <<, >>	Operadores AND, OR, XOR, NOT y de desplazamiento.
Instrucciones de estructuras de control	if, else, switch, while, for, do	Instrucciones selectivas y repetitivas
Valores numéricos	125, 3.1416, 6.12E02, 0x7F, 0157	Constantes numéricas enteras, reales, octales y hexadecimales

Tabla A.1: Algunos componentes léxicos y lexemas del lenguaje C.

Para comprender la necesidad de este módulo en el proceso de análisis debe tenerse en cuenta que un programa es una cadena de caracteres carente de estructura, que incluye múltiples sentencias, donde la apariencia de estas últimas es solo un efecto visual logrado mediante caracteres de control: <CR>, <LF>, <~~b~~> (espacio), <TAB>, etc. En efecto, los caracteres de control desplazan el cursor al comienzo de la línea (<CR>), lo avanzan a la siguiente línea (<LF>) o avanzan sobre la misma línea un espacio prefijado (<TAB>). El resultado que se obtiene tiene sentido estético para el observador, pero carece de significado para el compilador, que necesita reconocer los componentes de todas las sentencias que forman parte de un programa.

Como ejemplo, se presenta, en la Figura A.5, una sentencia almacenada en un archivo que toma la forma de una cadena de caracteres, luego la misma sentencia tal como se la visualiza en pantalla gracias a los caracteres de control y finalmente el detalle de los componentes del lenguaje identificado por el analizador léxico.

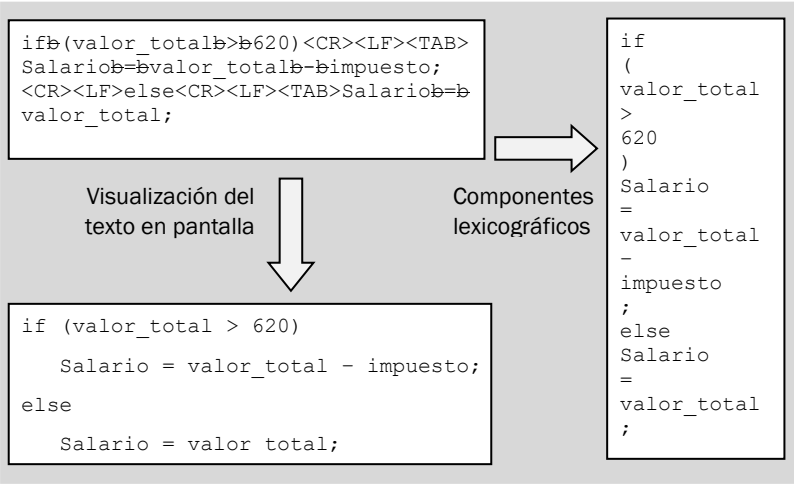

Figura A.5: Fragmento de un archivo de programa fuente (Lenguaje C), representación del mismo en pantalla y resultado de su procesamiento por un analizador léxico. Entre estos componentes léxicos se reconocen delimitadores, identificadores (variables y constantes), instrucciones, operadores, símbolos especiales y palabras reservadas, entre otros.

Se hace notar que en el Ejemplo 39 del Capítulo 3 se implementa un analizador léxico con un autómata finito con el fin de reconocer los lexemas que representan valores numéricos, es decir que forman parte de los componentes léxicos numéricos. Esto es posible porque las cadenas reconocidas son siempre elementos de un lenguaje regular, es decir, son generadas por una gramática regular y; por lo tanto, responden a expresiones regulares.

Aquí cabe acotar que, a partir de un archivo de especificaciones que describa las características léxicas de un lenguaje, se han propuesto herramientas destinadas a generar el código fuente de programas que deben ser compilados para ser utilizado como analizadores léxicos. Como ejemplos, pueden citarse Lex y Flex para la generación de analizadores léxicos de programas escritos en C y C++, así como JFlex y CookCC para hacer lo propio con Java.

Pasando ahora al aspecto operativo, debe observarse que, a medida que los componentes léxicos son identificados, se los clasifica, se los registra en la tabla de símbolos y se les asigna la dirección de la tabla de símbolos donde se almacenan los restantes atributos. La tabla de símbolos es una estructura que está destinada al almacenamiento y recuperación de todos los elementos que forman parte de un programa y cuyos registros contienen nombres y atributos. Inicialmente esta tabla es cargada por el analizador Léxico, luego se completa en las restantes fases de análisis y será empleada intensivamente en la siguiente etapa de síntesis.

Como ejemplo, la tabla de símbolos permite a través de sus atributos identificar un error en cualquier fase del proceso de compilación y relacionarlo con la línea del programa fuente en que se encuentra la sentencia causante del problema y el lexema involucrado.

Las técnicas y herramientas utilizadas en la construcción de estos analizadores léxicos son también de utilidad en otras áreas de la informática, tales como en lenguajes de consulta, editores de texto, sistemas de recuperación de información y procesamiento de lenguaje natural.

Análisis sintáctico

También se denomina análisis jerárquico y esta tarea está a cargo de los analizadores sintácticos. En esta fase, se reciben las secuencias de componentes léxicos que fueron identificadas en la fase anterior, y debido a que las combinaciones de estos componentes dan lugar a sentencias del programa fuente, es necesario comprobar que las sentencias sean sintácticamente correctas. Es decir, se debe verificar que todas las sentencias puedan haber sido generadas por la gramática del lenguaje fuente. En caso contrario, el analizador debe informar sobre los errores sintácticos detectados de manera precisa e inteligible.

Como ya es sabido, los lenguajes de programación (lenguajes fuente) son generados por gramáticas libres de contexto, y los lenguajes libres de contexto son reconocidos por los autómatas con pila. Por lo tanto, todo analizador sintáctico destinado a reconocer un lenguaje libre

de contexto tiene una capacidad computacional equivalente a la de un autómata con pila. Para esclarecer lo expuesto, es útil volver al ejemplo anterior, donde un analizador léxico permitió identificar los componentes del lenguaje y ahora el analizador sintáctico debe comprobar si la sentencia, mostrada en la Figura A.6 cumple las condiciones de la gramática del lenguaje C.

```
<if><(><valor_total><>><620><)><Salario><=><valor_total><->
<impuesto><;><else><Salario><=><valor_total><;>
```

Figura A.6: Sentencia a ser validada por el analizador sintáctico.

Aquí los componentes de la sentencia están encerrados entre el par de símbolos “<” y “>” con el fin de destacar que se trata de palabras indivisibles, que componen el *alfabeto de terminales* de la gramática.

Como ya fue expuesto anteriormente, la proliferación de nuevos lenguajes de programación y la necesidad de disponer de compiladores eficientes hizo que los analizadores sintácticos fueran muy estudiados, lo que dio lugar a numerosas variantes en su implementación. Sin embargo, todas ellas son esencialmente no deterministas y pueden agruparse en dos grandes tipos: los *descendentes* y los *ascendentes*.

Como fue explicado en el Capítulo 5, los Analizadores Sintácticos Descendentes (ASD), en inglés *Top-Down-Parser*, comienzan a operar a partir del axioma de la gramática y procuran desarrollar por izquierda el árbol de derivación sintáctica a medida que las sentencias son leídas. Si este proceso permite completar la lectura de las cadenas, significa que la sentencia responde a las reglas de producción de la gramática y es aceptada. Entre los analizadores descendentes, se encuentran los denominados “LL” (*Left-Left*), donde la primera “L” se refiere a que la cadena es leída de izquierda a derecha y la segunda a que el proceso corresponde al de una derivación por izquierda. En su forma básica, estos analizadores son esencialmente no deterministas y son llamados predictivos debido al proceso de búsqueda involucrado. Con el fin de mejorar el desempeño de los procesos de validación de las sentencias, se procuró disponer de analizadores LL deterministas, para los cual se propusieron numerosas variantes a través de procesos de preanálisis, llamados LL(k). El argumento *k* representa la cantidad de símbolos de la entrada que involucra el proceso de preanálisis, que implica la observación de símbolos de entrada posteriores al símbolo efectivamente leído.

Las gramáticas cuyos lenguajes pueden ser validados por analizadores LL(k) son conocidas como *gramáticas LL(k)* y cabe acotar que estos analizadores sintácticos fueron objeto de especial atención por parte de numerosos investigadores, entre ellos Niklaus Wirth. También hay que destacar que existen lenguajes independientes de contexto que no pueden ser validados por ningún analizador sintáctico LL(k), lo que estimuló el estudio del otro tipo de analizadores más complejos, que son los ascendentes.

Para superar la dificultad señalada la alternativa fueron los Analizadores Sintácticos Ascendentes (ASA), que también fueron tratados en el Capítulo 5. Los ASA comprueban la validez de las sentencias operando de abajo hacia arriba, es decir, construyen los árboles de derivación sintáctica desde las hojas hasta su raíz, que es el axioma de la gramática. En inglés, son denominados *Bottom Up Parsers* y, en general, estos analizadores disponen de mayor poder de reconocimiento que los analizadores descendentes.

Tal como ocurre con los analizadores descendentes, hay numerosas propuestas que responden a la concepción ascendente. Una de las principales es denominada LR, cuyo nombre se debe a que las cadenas son leídas de izquierda a derecha (*Left*) y se sigue un proceso de reducción por izquierda. A una reducción por izquierda, le corresponde una derivación por derecha y de allí proviene el segundo carácter de la denominación “LR” (*Left-Right*). El analizador sintáctico LR es también no determinista y su forma básica se ve muy beneficiada con el preanálisis, dando lugar a los denominados LR(k). Las gramáticas que dan lugar a lenguajes que pueden ser validados por analizadores LR(k) son reconocidas como *gramáticas LR(k)*

En resumen, los analizadores más avanzados recurren a algoritmos que resuelven de manera determinista un problema esencialmente no determinista, lo que permite implementar los analizadores LL(k) y LR(k) eliminando o reduciendo enormemente las costosas búsquedas que se

presentan en árboles sintácticos con factores de ramificación elevados. Con referencia a la comparación entre ambos, los analizadores LR(k) tienen mayor capacidad de validación que los LL(k), aunque existen lenguajes independientes de contexto que ningún analizador LR(k) puede tampoco reconocer. En el Capítulo 5, se brindan ejemplos de analizadores sintácticos descendentes y ascendentes, donde se comprueba que se tratan de autómatas no deterministas.

Por último, es necesario destacar que, al igual que en el caso de los analizadores léxicos, existe una amplia variedad de herramientas destinadas a generar analizadores sintácticos para gramáticas independientes de contexto. Entre otros, pueden citarse Byacc, Hyacc y Yacc para el lenguaje C, YooParse y Yacc++ para C++ y TPYacc para Turbo Pascal.

Análisis semántico

En el análisis semántico, se revisa al programa fuente, sintácticamente correcto, para reunir información sobre los tipos de las variables que será utilizada en la fase posterior de generación de código intermedio. Simultáneamente, se procuran identificar eventuales errores semánticos. Las comprobaciones en esta fase son denominadas estáticas (para distinguirlas de las comprobaciones realizadas en tiempo de ejecución, que son llamadas dinámicas) e incluyen la detección y comunicación de numerosos errores que corresponden a: *i*) comprobación de tipos, *ii*) comprobación de flujos de control, *iii*) comprobación de unicidad o coherencia en las denominaciones de identificadores, *iv*) coherencia en los argumentos de subprogramas y *v*) potenciales errores en tiempo de ejecución (variables no inicializadas, direccionamiento de arreglos fuera de límites, cocientes que pueden tomar valores nulos, etc.). Una de las formas más simples de implementar analizadores semánticos las brindan las gramáticas con atributos, que son estudiadas en el Capítulo 8.

Generación de código intermedio

La etapa de análisis prepara al programa fuente para ser convertido en un nuevo programa escrito en un lenguaje elemental que es normalmente denominado *lenguaje intermedio*. Tradicionalmente, fue utilizado el *código de tres direcciones*, cuyo nombre proviene del hecho que sus instrucciones incluyen tres registros, dos de ellos para operandos y el tercero para el resultado. Posteriormente, con la evolución de los lenguajes de alto nivel, se procuró normalizar este lenguaje intermedio y la tendencia fue adoptar un lenguaje universal denominado Uncol (***U****niversal* ***C****ompiler* ***O****riented* ***L****anguage*). De esta manera, se facilita la combinación de módulos destinados a interpretar diversos lenguajes fuentes (etapa de análisis) con los de la etapa de síntesis destinada a producir código para diversos lenguajes objeto. Esta normalización significa un inmenso ahorro en el esfuerzo de programación que requieren los compiladores de nuevos lenguajes y la posibilidad de acortar significativamente los tiempos requeridos para la presentación de nuevos productos y herramientas de programación.

Optimización de código

A pesar de su nombre, esta fase realiza una mejora en la calidad y eficiencia del código intermedio, pero difícilmente permita alcanzar un código óptimo. Para cumplir esta tarea, debe comenzarse por identificar las zonas críticas del programa, es decir, aquellas zonas que por ser ejecutadas reiteradamente afectan significativamente su comportamiento y donde una mejora se verá enormemente potenciada. Como ya se anticipó, es importante reconocer que en el proceso de optimización pueden perseguirse objetivos diferentes en la reducción de los recursos demandados, según la aplicación, rapidez de respuesta esperada, rapidez de los computadores, disponibilidad de memoria, etcétera.

Un aspecto a ser destacado es que, al efectuarse las mejoras requeridas por el proceso de optimización, se sigue un orden que es el siguiente: *i*) Mejoras que se refieren a la calidad de la implementación del programa desde el punto de vista lógico y *ii*) mejoras particulares para el mejor aprovechamiento global de una cierta máquina, como son la selección de las instrucciones más apropiadas, direccionamientos, etcétera.

Generación del programa objeto

Es la fase final del proceso de compilación, en la que se toma como entrada a la representación intermedia y se produce un programa objeto equivalente, que debe ser correcto, eficiente, apropiado a la máquina en la que se va a operar y apto para dar lugar a un ejecutable compatible con el entorno (sistema operativo). Algunas de las tareas de esta fase son: *i*) selección del orden de evaluación de expresiones, *ii*) selección de instrucciones de la máquina de destino, *iii*) asignación de registros, memoria y otros recursos y *iv*) administración de la memoria para datos y programa.

Gestor de la tabla de símbolos

El gestor de la tabla de símbolos es en realidad un administrador de una base de datos que contiene los identificadores del programa fuente y sus atributos. En las fases de análisis, esta base es definida y completada, para servir luego de consulta en las fases de síntesis. Esta tabla contiene un registro por cada identificador y es cargada a medida que el analizador léxico detecta nuevos elementos en el programa fuente. Posteriormente, los restantes atributos de cada identificador son definidos en las fases de análisis sintáctico y semántico. Entre los atributos pueden mencionarse los siguientes: *i*) identidad (constante, variable, arreglo, función, rótulo, etc.), *ii*) tipo (entero, real, carácter, cadena, etc.), *iii*) dimensiones (para el caso de un arreglo), *iv*) argumentos (para el caso de función o subprograma) y *v*) dirección de memoria asignada.

Identificación y gestión de errores

El gestor de errores, esencialmente activo en las fases de análisis, detecta los errores, los asocia a determinada línea del programa fuente e intenta su recuperación con la finalidad de proseguir con la tarea de compilación. Para esta última tarea, el registrador dispone de un corrector lexicográfico, un corrector sintáctico y un corrector semántico, que operan según la fase en la que fue reportada la irregularidad.

Cuando los compiladores están incorporados a Ambientes Integrados de Desarrollo (IDE) los errores son mostrados sobre el editor del programa fuente, facilitando la identificación y corrección por parte del programador. Antiguamente, se generaban listados para ser impresos con todos los detalles de los errores del programa.

Errores de programación

Al evaluarse la calidad de un producto software se lo verifica y valida. Como fue anticipado, el proceso de verificación tiene por finalidad la comprobación de que el producto ha sido construido correctamente, mientras que el objetivo de la validación es comprobar que se construyó el producto correcto, es decir que responde a las especificaciones planteadas en los requerimientos. Al hablarse de errores, se hará aquí referencia a aquellos que se incorporaron en la programación e integración del producto, que deben ser identificados en el proceso de verificación, y que en caso de no hacerse se convertirán en defectos, es decir, condiciones potenciales de falla. Los demás errores, es decir aquellos que puedan ser detectados en los procesos de validación, fueron introducidos en las etapas de análisis y diseño del sistema y no son aquí considerados. Para detectar ambos tipos de errores, los procesos de desarrollo incluyen una etapa denominada *testing*.

Como se advierte con facilidad, el proceso de verificación, que incluye la detección de errores y depuración de programas se verá enormemente beneficiado por las cualidades del compilador en particular y del IDE en general. El primero por advertir los errores incorporados en el código y el segundo por suministrar recursos para posibilitar el seguimiento de las condiciones de falla remanentes y estudiar en profundidad su contexto, hasta identificar sus causas y resolverlas.

A continuación, se enumeran las clases de errores con las que se enfrenta un programador y luego se las describe en mayor detalle.

Errores de programación
- lexicográficos
- sintácticos
- semánticos
- por falla del compilador
- de ejecución

Las primeras tres clases de errores están al alcance de ser identificadas por las fases correspondientes de la etapa de análisis del compilador, formando parte de lo que se denomina *evaluación estática*. La última clase requiere la ejecución del programa y será más efectiva en la medida en que estén mejor planeados los casos de prueba, ya que deberían procurarse recrear todas las secuencias lógicas presentes. Esto se llama *evaluación dinámica*. Aquí debe tenerse en cuenta que solo es posible asegurar la corrección de los programas a partir de pruebas sobre todas las entradas y condiciones de operación posibles, lo que puede dar lugar a un enorme esfuerzo que en la práctica es inviable. A partir de reconocer este problema, que puede ser resumido puntualizando que la *verificación dinámica* permite identificar la presencia de errores pero nunca asegurar la ausencia de ellos, se pone claramente de manifiesto la gran importancia de las cualidades del compilador para contribuir a una *evaluación estática* tan detallada y completa como sea posible. Naturalmente, siempre habrá errores remanentes que deben ser detectados en tiempo de ejecución y el esfuerzo que estas pruebas demandarán puede ser anticipado a partir de la "complejidad ciclomática" de los módulos, que es un indicador propuesto por McCabe para evaluar la cantidad de caminos linealmente independientes presentes en una pieza de código.

Las posibles reacciones del compilador frente a la detección de un error en el código fuente dependen del tipo de compilador, siendo las más importantes las siguientes:

a) Genera el programa objeto sin advertir el error.

b) Detecta el primer error, lo señala y suspende la operación.

c) Detecta un error e intenta sin éxito su recuperación, para ingresar en un lazo infinito o terminar el proceso anticipadamente.

d) A partir del primer error, suspende la generación del código objeto, pero continúa el análisis y la búsqueda de otros posibles errores.

e) Detecta un error y lo recupera con éxito, para proseguir la compilación, y procurar completar normalmente la generación del programa objeto.

La mayoría de los compiladores modernos se comportan según lo señalado en "d" y "e", dependiendo en última instancia del tipo de error encontrado.

Volviendo a la clasificación de los errores, el significado y alcance de cada una es el siguiente.

Errores lexicográficos

Son errores identificados en la fase de análisis lexicográfico y corresponden a los siguientes casos:

a) Presencia de un símbolo que pertenece al alfabeto de terminales, pero que no puede formar parte de la cadena que está siendo reconocida. Ejemplos: operadores aritméticos en medio de una cadena, prefijo numérico en una cadena alfanumérica, símbolos alfabéticos u operadores en una cadena que representa un valor numérico, etcétera.

b) Cadena de comentarios incorrectamente delimitada. Ejemplos: falta del separador de inicio o fin de un comentario.

c) Errores en palabras reservadas por omisión o ubicación incorrecta de caracteres.

d) Componente léxico truncado como consecuencia de final inesperado del archivo fuente.

Errores sintácticos

Son errores de estructura que se originan en el incumplimiento de alguna de las reglas de producción de la gramática y corresponden a los siguientes casos principales:

a) Cadenas de paréntesis o corchetes que no están balanceadas o incorrectamente ubicados.

b) Cadenas no permitidas en estructuras sintácticas como consecuencia de la ausencia de operadores u operandos.

c) Estructuras sintácticas alteradas por exceso o ausencia de delimitadores o separadores.

d) Palabras reservadas mal formadas o usadas indebidamente.

e) Bloques indebidamente delimitados por estar mal balanceados.

En el caso del análisis sintáctico, se ha hecho un importante esfuerzo en definir estrategias de recuperación de errores. Sin embargo, la mayoría de las estrategias son heurísticas y carecen de validez general. Las principales técnicas son las siguientes:

i) *Recuperación en modo de pánico*: luego de que un error es detectado, el analizador avanza hasta completar la sentencia para volver a sincronizarse y poder reanudar el proceso.

ii) *Recuperación por reemplazo*: se realizan cambios en el código para intentar superar los errores detectados en forma temporaria y luego informarlos.

iii) *Recuperación por expansión de la gramática*: se introducen producciones especiales en la gramática para superar ciertos errores. Como en el caso anterior, el objetivo es completar el proceso de compilación para informar todos los errores una vez finalizado.

Errores semánticos

Estos errores son detectados en la fase de análisis semántico y los principales tienen que ver con:

a) Identificadores no declarados.

b) Declaraciones múltiples de un mismo identificador.

c) Incompatibilidad de tipos entre operadores y operandos.

d) Incompatibilidad de tipos entre argumentos formales y reales en subprogramas.

e) Límites inválidos en instrucciones de control de flujo.

f) Errores potenciales en tiempo de ejecución (variables no inicializadas, direccionamiento de arreglos fuera de sus límites, cocientes con posibles denominadores nulos, etcétera).

Fallas en el compilador

Se trata de errores que se presentan en tiempo de compilación, pero no tienen que ver con defectos en el programa fuente sino con haber excedido ciertos límites específicos referidos a la capacidad del compilador. Algunos de los límites que pueden ser excedidos son los siguientes:

a) Tamaño máximo de la tabla de símbolos.

b) Límite de cantidad de bloques en un programa.

c) Limite en el número de anidamientos de ejecuciones repetidas.

d) Encadenamiento de llamadas a funciones.

e) Límite en el tamaño de las pilas en el análisis sintáctico o semántico.

Errores de ejecución

Estos errores se manifiestan durante la ejecución del programa o lo hacen a través de la entrega de resultados erróneos. En el primer caso, se ponen en evidencia por la súbita interrupción del proceso, normalmente acompañados de un mensaje informativo, mientras que en el segundo caso los errores son más peligrosos, ya que durante el proceso pasan inadvertidos, y sus consecuencias pueden tener fuerte impacto, imposible de precisar. Pueden citarse como ejemplo el cálculo incorrecto de importes a ser pagados en un sistema de liquidación de sueldos o la producción de un medicamento defectuoso por un error en el dosificador se sus componentes.

Los errores de ejecución tienen su origen en acciones equivocadas u omisiones del programador al momento de escribir el programa fuente y responden a alguna de las siguientes causas: *i*) errores de lógica (ejemplo: ingreso en un ciclo infinito), *ii*) inaccesibilidad de periféricos de E/S (ejemplo: intento de operar sobre un archivo inaccesible o protegido, no habiendo verificado previamente su disponibilidad), *iii*) errores en el código (ejemplo: nombre de variables

cambiadas) y *iv*) imprevisiones (ejemplo: ausencia de verificaciones que eviten el procesamiento de datos erróneos)

Los errores indicados en "*i*" y "*ii*" se pondrán claramente de manifiesto por provocar un final inesperado en la ejecución, que en general es acompañado del correspondiente mensaje indicando la causa de la falla y la dirección de la variable o procedimiento involucrado. Por el contrario, los errores incluidos en "*iii*" y "*iv*" pueden muchas veces pasar inadvertidos y afectar los resultados sin ninguna otra manifestación aparente, lo que los torna muy peligrosos.

Consecuencias de los errores

La importancia del tratamiento de los errores justifica entrar en más detalles, aún a riesgo de reiterar algunos de los conceptos ya expresados anteriormente. En este caso, se agruparán los errores según el momento en que se producen y se pondrá el foco en sus consecuencias, lo que permite reconocer la importancia de los esfuerzos orientados a reducir la posibilidad de cometer errores de programación.

Errores en tiempo de compilación

Normalmente, sus consecuencias están circunscriptas al ámbito de desarrollo y de programación de los sistemas. Estos errores afectan los rendimientos y conducen a mayores plazos de programación y a mayores costos. Sin embargo, si estos errores son apropiadamente corregidos no trascienden al usuario ni afectarán la operación del sistema.

Errores que interrumpen la ejecución

Crean incertidumbre y dudas por trascender al ámbito del usuario. Las consecuencias de este tipo de fallas podrán ser más o menos severas dependiendo de la aplicación, pudiendo provocar desde pérdida de información hasta el descontrol o detención de equipos automáticos.

Las fallas pueden ser, a su vez, sistemáticas, por repetirse ante ciertas condiciones de operación, o pseudo-aleatorias. En este último caso, las manifestaciones no parecen guardar relación entre sí, aunque es muy probable que se trate de una falla sistemática originada en una combinación compleja de condiciones, muy difíciles de advertir. En efecto, la mayoría de las fallas que tienen apariencia aleatoria son en realidad sistemáticas, lo que recién se comprueba con un análisis minucioso que permita identificar sus causas.

Las fallas verdaderamente aleatorias son aquellas que no se producen bajo circunstancias específicas y pueden darse en cualquier etapa de un proceso. Si se admite que el software es determinista, es decir cumple acciones predeterminadas, no debería nunca esperarse una falla aleatoria.

Errores que afectan los resultados

La posibilidad de que como consecuencia de errores de programación un sistema entregue resultados incorrectos y que esto pase inadvertido es altamente preocupante. Las consecuencias pueden tener una gravedad insospechada, ya que debe tenerse en cuenta que en la actualidad los programas de computación están presentes en todos los campos de la actividad humana, involucrando equipos de diagnóstico, teléfonos celulares, instrumental de laboratorios, equipos industriales, máquinas dosificadoras y envasadoras, procesamiento de movimientos y saldos de cuentas bancarias, etc. Por ello, debe considerarse la responsabilidad legal del autor del software ante este tipo de fallas.

Evolución y tendencias de los lenguajes de programación

Los lenguajes de programación y sus compiladores han venido evolucionado a partir de un equilibrio entre las necesidades del mercado, la potencialidad de los equipos de computación disponibles en cada época y los continuos progresos experimentados en el campo de la ciencia de la computación. A partir de esta consideración, pueden distinguirse dos épocas claramente diferentes:

La primera época llega hasta mediados de los años ochenta y se caracteriza por la necesidad de hacer posible las aplicaciones en un marco de escasez de los recursos de hardware: escasa

memoria o memoria segmentada, discos rígidos de poca capacidad y periféricos de E/S lentos, entre otras.

La segunda incluye la época actual y el futuro cercano. Los recursos de hardware son suficientemente holgados para la mayoría de las aplicaciones convencionales y experimentan un proceso de continua mejora. En ese contexto, el esfuerzo se orienta en las siguientes direcciones: *i*) mejorar la eficiencia y aumentar la productividad de programadores, *ii*) aumentar la calidad de los productos software y *iii*) facilitar el futuro mantenimiento de los sistemas.

En este promisorio contexto, no puede dejar de mencionarse un aspecto poco favorable, que es la creciente demanda de recursos de los sistemas operativos y demás software de base, que dejan algunas dudas en cuanto a sus indicadores de calidad y que distraen recursos destinados a ofrecer funcionalidades que no son prioritarios ni demandados por el usuario, sino más bien impuestas por sus autores en un mercado claramente oligopólico. Naturalmente, este es un tema polémico que no tiene una respuesta concluyente.

Comentario final referido al tratamiento de los compiladores

Tal como fue anticipado, se consideró necesario incluir este apéndice con el fin de ofrecer una presentación conjunta de conceptos generales referidos a compiladores. Detalles específicos sobre los procesos asociados al análisis léxico, sintáctico y semántico se brindan en los capítulos correspondientes, en los que se incluyen ejemplos y ejercicios. Para tratar este tema con aún mayor profundidad, se sugiere remitirse a la excelente bibliografía disponible sobre el tema, alguna de la cual se cita en las referencias del libro.

Para cerrar este apéndice, se transcribe, con algunas leves variantes, un párrafo del prefacio del libro *Compiladores* de Aho, Sethi y Ullman que se considera muy oportuno para los propósitos de este texto: "*Aunque es probable que poca gente construya o llegue a dar mantenimiento a un compilador… el lector puede aplicar con provecho las ideas y técnicas sobre compiladores al diseño de software convencional"*.