Capítulo 1

El estudio de los lenguajes de programación

Cualquier notación para describir algoritmos y estructuras de datos se puede calificar como un lenguaje de programación; sin embargo, en este libro nos interesan principalmente los que se implementan en una computadora. El sentido en el cual se puede "implementar" un lenguaje se considera en los dos capítulos siguientes. En el resto de la parte I se estudia con detalle el diseño e implementación de los diversos componentes de un lenguaje. El objetivo es examinar las características de un lenguaje, independientes de cualquier lenguaje particular, y mostrar ejemplos de una amplia clase de lenguajes de uso común.

En la parte II de este libro, se ilustra la aplicación de estos conceptos en el diseño de nueve lenguajes de programación importantes y sus dialectos: Ada, C, C++, FORTRAN, LISP, ML, Pascal, Prolog y Smalltalk. Además, también se ofrecen breves síntesis acerca de otros lenguajes que han tenido un impacto en este campo. Esta lista incluye APL, BASIC, COBOL, Forth, PL/I, y SNOBOL4. Sin embargo, antes de abocarnos al estudio general de los lenguajes de programación, merece la pena entender por qué es valioso un estudio de esta naturaleza para el programador de computadoras.

1.1 ¿POR QUÉ ESTUDIAR LENGUAJES DE PROGRAMACIÓN?

Se han diseñado e implementado cientos de lenguajes de programación distintos. Ya en 1969, [SAMMET 1969] elaboró una lista con 120 de uso bastante amplio, y desde entonces se han desarrollado muchos más. Sin embargo, casi ningún programador se aventura a usar más de unos cuantos lenguajes, y muchos restringen por completo su programación a uno o dos. De hecho, los programadores profesionales suelen trabajar en instalaciones de cómputo donde se requiere el uso de un lenguaje particular, como C, Ada o FORTRAN. ¿Cuál es entonces la ventaja de estudiar una variedad de lenguajes diferentes que es poco probable que uno llegue a utilizar?

Existen razones excelentes para un estudio de esta naturaleza, siempre y cuando se profundice más allá de la consideración superficial de las "características" de los lenguajes y se exploren los conceptos de diseño subyacentes y su efecto sobre la implementación de lenguajes. Seis razones primordiales vienen de inmediato a la mente:

1. *Mejorar la habilidad para desarrollar algoritmos eficaces.* Muchos lenguajes incluyen características que, cuando se usan en forma apropiada, benefician al programador pero que, si se usan incorrectamente pueden desperdiciar grandes cantidades de tiempo de cómputo o conducir al programador a errores lógicos que consumen mucho tiempo. Incluso un programador que ha utilizado un lenguaje durante años puede no entender todas sus características. Un ejemplo típico es la *recursión*, una característica de programación que, cuando se emplea correctamente, permite la implementación directa de algoritmos elegantes y eficientes. Pero si no se usa de manera apropiada, puede ocasionar que el tiempo de ejecución aumente de manera astronómica. Es probable que el programador que desconoce las cuestiones de diseño y las dificultades de implementación que la recursión implica, rehuya esta construcción un poco misteriosa. Sin embargo, un conocimiento básico de sus principios y técnicas de implementación permite al programador comprender el costo relativo de la recursión en un lenguaje particular y, a partir de esta comprensión, determinar si una situación particular de programación amerita su uso.

   Constantemente se están presentando nuevos métodos de programación en la bibliografía de la materia. El uso óptimo de conceptos como programación orientada a objetos, programación lógica o programación concurrente, por ejemplo, requiere una comprensión de los lenguajes que implementan estos conceptos.

2. *Mejorar el uso del lenguaje de programación disponible.* A través del entendimiento de cómo se implementan las características del lenguaje que uno usa, se mejora grandemente la habilidad para escribir programas más eficientes. Por ejemplo, cuando se entiende cómo crea y manipula el lenguaje datos como arreglos, cadenas, listas o registros, se conocen los detalles de implementación de la recursión o se comprende cómo se construyen clases de objetos permite construir programas más eficientes integrados con tales componentes.

3. *Acrecentar el propio vocabulario con construcciones útiles sobre programación.* El lenguaje sirve a la vez como una ayuda y como una restricción para el pensamiento. Las personas usan el lenguaje para expresar pensamientos, pero también sirve para estructurar la manera como uno piensa, en la medida en que es difícil pensar en formas que no permiten una expresión directa en palabras. La familiaridad con un único lenguaje de programación tiende a tener un efecto similar de restricción. Al buscar datos y estructuras de programa adecuados para la solución de un problema, uno tiende a pensar sólo en estructuras que son susceptibles de expresión inmediata en lenguajes con los que uno está familiarizado. A través del estudio de las construcciones suministrados por una amplia gama de lenguajes, así como de la manera en que se aplican estas construcciones, el programador aumenta su "vocabulario" de programación. La comprensión de las técnicas de implementación es particularmente importante porque, para usar una construcción cuando se está programando en un lenguaje que no la provee directamente, el programador debe aportar su propia implementación de la construcción en términos de los elementos primitivos que efectivamente proporciona el lenguaje. Por ejemplo, la estructura de control de subprogramas que se conoce como *corrutinas* es útil en muchos programas, pero pocos lenguajes proporcionan una función de corrutinas

directamente. Un programador en C o en FORTRAN, sin embargo, puede diseñar fácilmente un programa que utilice una estructura de corrutinas, para luego implementarlas como programas en C o FORTRAN si está familiarizado con el concepto de corrutina y con su implementación, y al hacerlo puede ser capaz de suministrar precisamente la estructura de control correcta para un programa grande.

4. *Hacer posible una mejor elección del lenguaje de programación.* Cuando se presenta la situación, un conocimiento de diversos lenguajes puede permitir la elección de un lenguaje que sea precisamente el adecuado para un proyecto particular, con lo cual se reduce el esfuerzo de codificación requerido. Por ejemplo, los lenguajes que se describen en la parte II de este libro se proyectaron para distintas aplicaciones. Los usos que requieren cálculos numéricos se pueden diseñar fácilmente en lenguajes como C, FORTRAN o Ada. El desarrollo de aplicaciones útiles en la toma de decisiones, como en tareas de inteligencia artificial, se podría escribir con más facilidad en LISP, ML o Prolog. El conocimiento de las características básicas de las fortalezas y debilidades de cada lenguaje proporciona al programador una selección más amplia de alternativas.

5. *Facilitar el aprendizaje de un nuevo lenguaje.* Un lingüista, a través de una comprensión profunda de la estructura subyacente de los lenguajes naturales, suele poder aprender un nuevo idioma con más rapidez y facilidad que el novato esforzado que entiende poco de la estructura de incluso su lengua nativa. De manera similar, un conocimiento concienzudo de diversas construcciones de lenguajes de programación y técnicas de ejecución permite al programador aprender un nuevo lenguaje de programación con más facilidad cuando ello es necesario.

6. *Facilitar el diseño de un nuevo lenguaje.* Pocos programadores piensan alguna vez de sí mismos como diseñadores de lenguajes; sin embargo, todo programa tiene una *interfaz de usuario* que es, de hecho, una forma de lenguaje de programación. La interfaz de usuario se compone de los comandos y formatos de datos que se suministran para que el usuario se comunique con el programa. El diseñador de la interfaz de usuario para un programa grande como un editor de texto, un sistema operativo o un paquete de gráficos debe ocuparse de muchas de las mismas cuestiones que están presentes en el diseño de un lenguaje de programación para usos generales. Este aspecto del diseño de programas se suele simplificar si el programador está familiarizado con diversas construcciones y métodos de implementación de lenguajes de programación ordinarios.

Debe resultar evidente que el estudio de los lenguajes de programación implica mucho más que un simple vistazo superficial a sus características. De hecho, muchas similitudes en cuanto a peculiaridades son engañosas. La misma característica en dos lenguajes diferentes se puede implementar de dos maneras muy distintas y, de esta manera, las dos versiones pueden diferir considerablemente en cuanto al costo de uso. Por ejemplo, casi todos los lenguajes proveen una operación de adición como una primitiva, pero el costo de ejecutar una adición en C, COBOL o Smalltalk puede variar en un orden de magnitud. El estudio de los lenguajes de programación deberá incluir por necesidad el estudio de técnicas de implementación, en particular técnicas para la representación del tiempo de ejecución de diferentes construcciones.

En la parte I de este libro, se analizan numerosas construcciones de lenguaje acompañadas en casi todos los casos de uno o más diseños para la implementación de la construcción en una computadora convencional. Sin embargo, no se ha hecho intento alguno por ser exhaustivo en la cobertura de todos los métodos de implementación posibles. El mismo lenguaje o construcción, si se implementa en la computadora local del lector, puede diferir en forma radical en cuanto a costo o a detalle de estructura cuando se han utilizado diferentes técnicas de implementación o cuando el equipo físico subyacente de la computadora difiere de la estructura convencional sencilla que aquí se supone.

1.2 BREVE HISTORIA DE LOS LENGUAJES DE PROGRAMACIÓN

Los diseños y métodos de implementación de lenguajes de programación han evolucionado de manera continua desde que aparecieron los primeros lenguajes de alto nivel en la década de 1950. De los nueve lenguajes que se describen en la parte II, el diseño de las primeras versiones se hizo durante los años cincuenta; Ada, C, Pascal, Prolog y Smalltalk datan de los años setenta, y C++ y ML de los años ochenta. En las décadas de 1960 y 1970, se solían desarrollar nuevos lenguajes como parte de proyectos importantes de desarrollo de software. Cuando el Departamento de Defensa de Estados Unidos realizó un estudio como parte de sus esfuerzos preparatorios para el desarrollo de Ada en los años setenta, encontró que se estaban usando más de 500 lenguajes en diversos proyectos de defensa.

1.2.1 Desarrollo de los primeros lenguajes

En las páginas que siguen se sintetiza en forma breve el desarrollo de lenguajes durante los primeros días de la computación, en general desde mediados de los años cincuenta hasta principios de los setenta. Los desarrollos más recientes se incluyen como parte del estudio de los diversos lenguajes en la parte II de este libro.

Lenguajes basados en el cálculo numérico. La tecnología de computadoras más antigua data de la época justo antes de la Segunda Guerra Mundial, de finales de los años treinta a principios de los cuarenta. Estas primeras máquinas estaban proyectadas para resolver problemas numéricos y se pensaba en ellas como en *calculadoras electrónicas*. No debe sorprender, por tanto, que los cálculos numéricos hayan sido la forma dominante de aplicación para estas máquinas iniciales.

A principios de los años cincuenta comenzaron a aparecer notaciones simbólicas. Grace Hopper encabezó un grupo en Univac para desarrollar el lenguaje A-0, y John Backus desarrolló Speedcoding para la IBM 701. Ambos se proyectaron para compilar expresiones aritméticas sencillas en un lenguaje de máquina ejecutable.

El verdadero avance tuvo lugar de 1955 a 1957, cuando Backus encabezó un equipo para desarrollar FORTRAN, o FORmula TRANslator (traductor de fórmulas). Como en el caso de los primeros esfuerzos, los datos de FORTRAN se orientaban en torno a cálculos numéricos, pero la meta era un lenguaje de programación en toda forma que incluyera estructuras de control, condicionales y enunciados de entrada y de salida. Puesto que pocos pensaban que el lenguaje resultante pudiera competir con el lenguaje ensamblador codificado a mano, se

concentraron todos los esfuerzos en su ejecución eficiente y se diseñaron diversos enunciados específicamente para la IBM 704. Los conceptos como la ramificación aritmética de tres vías de FORTRAN provinieron directamente del equipo físico (hardware) de la 704, y los enunciados como *READ INPUT TAPE* (leer cinta de entrada) suenan curiosos en la actualidad. No era muy elegante, pero en aquellos días poco se sabía acerca de la programación "elegante" y el lenguaje era rápido para el hardware que se tenía.

FORTRAN tuvo un éxito extraordinario —tanto así que cambió para siempre la programación y probablemente preparó el escenario para su reemplazo final por otros lenguajes—. Se hizo una revisión de FORTRAN en 1958 (FORTRAN II) y otra más unos cuantos años después (FORTRAN IV). Casi todos los fabricantes implementaron una versión del lenguaje y reinaba el caos. Finalmente, en 1966, FORTRAN IV se convirtió en estándar con el nombre de FORTRAN 66 y ha sido actualizado dos veces desde entonces, a FORTRAN 77 y FORTRAN 90. Sin embargo, el número extremadamente grande de programas escritos en estos primeros dialectos ha sido causa de que las generaciones subsiguientes de traductores sean casi todas compatibles con estos antiguos programas e inhibe el uso de características modernas de programación.

A causa del éxito de FORTRAN, existía el temor, en especial en Europa, de que IBM dominara la industria. La GAMM (la Sociedad Alemana de Matemáticas Aplicadas) organizó un comité para diseñar un lenguaje universal. En Estados Unidos, la Association for Computing Machinery (ACM; asociación para maquinaria de cómputo) también organizó un comité similar. Aunque inicialmente los europeos temían ser dominados por los estadounidenses, los comités se fusionaron. Bajo el liderazgo de Peter Naur, el comité desarrolló el International Algorithmic Language (IAL; lenguaje algorítmico internacional). Aunque se propuso ALGOrithmic Language (ALGOL; lenguaje algorítmico), el nombre no fue aprobado. Sin embargo, el uso común obligó a cambiar el nombre oficial y el lenguaje se conoció como ALGOL 58. Una revisión tuvo lugar en 1960 y el ALGOL 60 (con una revisión menor en 1962) se convirtió en el lenguaje "académico" de cómputo estándar desde los años sesenta hasta principios de los setenta.

En tanto que una meta para el FORTRAN era la eficiencia en la IBM 704, ALGOL tenía objetivos muy distintos:

1. La notación de ALGOL debería aproximarse a las matemáticas normales.
2. ALGOL debería ser útil para la descripción de algoritmos.
3. Los programas en ALGOL deberían ser compilables a lenguaje de máquina.
4. ALGOL no debería estar ligado a una sola arquitectura de computadora.

Éstos resultaron ser objetivos muy ambiciosos para 1957. Para dar cabida a la independencia de las máquinas, no se incluyó entrada ni salida en el lenguaje; se podían escribir procedimientos especiales para estas operaciones. Si bien esto hizo a los programas ciertamente independientes de un hardware particular, también significaba que cada implementación sería incompatible por necesidad con otra. Con el propósito de mantenerse cerca de las matemáticas "puras", los programas se consideraban como sustituciones de macros, lo cual condujo al concepto de paso de parámetros de *llamada por nombre* y, como se verá en la sección 7.3.1, es extremadamente difícil implementar bien la llamada por nombre.

ALGOL nunca alcanzó éxito comercial en Estados Unidos, aunque consiguió cierto éxito en Europa. Sin embargo, tuvo un impacto más allá de su uso. Como ejemplo de ello, Jules Schwarz de SDC desarrolló una versión de IAL (Jules' Own Version of IAL, o JOVIAL) que se convirtió en una norma para aplicaciones de la Fuerza Aérea de Estados Unidos.

Backus fue el editor del informe de ALGOL que definía el lenguaje [BACKUS 1960]. Usó una notación sintáctica comparable con el concepto de lenguaje libre de contexto desarrollado por Chomsky [CHOMSKY 1959]. Esto constituyó la introducción de la teoría formal de la gramática al mundo de los lenguajes de programación (sección 3.3). A causa de su propio papel y el de Naur en el desarrollo de ALGOL, la notación se conoce ahora como BNF, o Backus Naur Form (forma de Backus Naur).

Como un ejemplo más de la influencia de ALGOL, Burroughs, un vendedor de computadoras que, más tarde, se fusionó con Sperry Univac para formar Unisys, descubrió la obra de un matemático polaco llamado Lukasiewicz. Él había desarrollado una técnica interesante que permitía escribir expresiones matemáticas sin paréntesis, con un proceso eficiente de evaluación con base en pilas. Aunque no se trataba de un resultado matemático importante, esta técnica tuvo un efecto profundo sobre la teoría de compiladores. A través del uso de métodos basados en la técnica de Lukasiewicz, Burroughs desarrolló el hardware de computadora B5500 con base en una arquitectura de pilas y pronto tuvo un compilador de ALGOL mucho más rápido que cualquier compilador de FORTRAN.

En este punto la historia comienza a divergir. El concepto de tipos definidos por el usuario se desarrolló en los años sesenta, y ni FORTRAN ni ALGOL disponían de esta clase de características. Simula-67, desarrollado por Nygaard y Dahl de Noruega, introdujo el concepto de clases en ALGOL. Esto le proporcionó a Stroustrup la idea para sus clases de C++ (sección 12.2) como una extensión de C más tarde en los años ochenta. Wirth desarrollo ALGOL-W a mediados de la década de 1960 como una extensión de ALGOL. Este diseño tuvo sólo un éxito menor; sin embargo, entre 1968 y 1970 Wirth desarrolló Pascal (capítulo 11), el cual se convirtió en *el* lenguaje de la ciencia de la computación en los años setenta. Otro comité intentó repetir el éxito de ALGOL 60 con ALGOL 68, pero el lenguaje era radicalmente distinto y demasiado complejo para que la mayoría lo pudiera entender o implementar de manera eficaz.

Con la introducción de su nueva línea de computadoras 360 en 1963, la IBM desarrolló NPL (New Programming Language; nuevo lenguaje de programación) en su Laboratorio Hursley en Inglaterra. Después de algunas quejas por parte del English National Physical Laboratory (Laboratorio de Física Nacional Inglés), el nombre se cambió a MPPL (Multi-Purpose Programming Language; lenguaje de programación de usos múltiples) y más tarde se acortó a sólo PL/I. El PL/I fusionó los atributos numéricos de FORTRAN con las características de programación para negocios de COBOL. PL/I alcanzó un éxito moderado en los años setenta (fue uno de los lenguajes que se incluyeron en la segunda edición de este libro), pero su uso actual está disminuyendo al ser reemplazado por C y Ada (capítulo 12). El subconjunto educativo PL/C logró un éxito modesto en los años setenta como compilador estudiantil de PL/I (página 80). El BASIC (página 71) se desarrolló para satisfacer las necesidades de cálculo numérico del no científico, pero se ha extendido mucho más allá de su meta original.

Lenguajes para negocios. El procesamiento de datos de negocios era uno de los primeros dominios de aplicación por desarrollar después de los cálculos numéricos. Grace Hopper

encabezó un grupo en Univac para desarrollar FLOWMATIC en 1955. La meta era desarrollar aplicaciones de negocios usando una forma de texto parecido al inglés.

En 1959, el Departamento de Defensa de Estados Unidos. patrocinó una reunión para desarrollar el Common Business Language (CBL; lenguaje común para negocios), el cual habría de ser un lenguaje orientado a negocios que usara el inglés tanto como fuera posible para su notación. A causa de la divergencia en las actividades de muchas compañías, se formó un Comité de Corto Alcance para desarrollar rápidamente el lenguaje. Aunque pensaban que estaban proyectando un lenguaje provisional, las especificaciones, publicadas en 1960, eran los diseños para el COBOL (COmmon Business Oriented Language; lenguaje común orientado a negocios). COBOL fue revisado en 1961 y 1962, estandarizado en 1968 y revisado una vez más en 1974 y 1978. (Comentarios adicionales en la página 272.)

Lenguajes para inteligencia artificial. El interés en los lenguajes para inteligencia artificial se inició en los años cincuenta con el IPL (Information Processing Language; lenguaje de procesamiento de información) de la Rand Corporation. El IPL-V se conocía en forma bastante amplia, pero su uso estaba limitado por su diseño de bajo nivel. El avance importante ocurrió cuando John McCarthy, del MIT, diseñó LISP (LISt Processing; procesamiento de listas) para la IBM 704. LISP 1.5 se convirtió en la implementación "estándar" de LISP durante muchos años. Más recientemente, Scheme y Common LISP han continuado esa evolución (Sección 13.1.

LISP fue proyectado como un lenguaje funcional para procesamiento de listas. El dominio ordinario de problemas para LISP comprendía la búsqueda. Los juegos eran un campo de pruebas natural para LISP, puesto que el programa usual de LISP desarrollaba un árbol de posibles movimientos (como una lista enlazada) y luego recorría el árbol en busca de la estrategia óptima. Un paradigma alternativo era el procesamiento de cadenas, donde la solución habitual implicaba la transformación de texto de un formato a otro. La traducción automática de máquina, donde se podían sustituir cadenas de símbolos por otras cadenas, era el dominio natural de aplicación. El COMIT, de Yngve del MIT, fue uno de los primeros lenguajes en este dominio. Cada enunciado de programa era muy parecido a una producción libre de contexto (sección 3.3.1) y representaba el conjunto de sustituciones que se podían hacer si se encontraba esa cadena en los datos. Como Yngve mantuvo patentado su código, un grupo de los Laboratorios Bell de AT&T decidió desarrollar su propio lenguaje, lo cual dio por resultado el SNOBOL (página 255).

En tanto que LISP fue proyectado para aplicaciones de procesamiento de listas para usos generales, Prolog (sección 14.1) fue un lenguaje para usos especiales cuya estructura básica de control y su estrategia de implementación se basaban en conceptos de lógica matemática.

Lenguajes para sistemas. A causa de la necesidad de eficiencia, el uso de lenguaje ensamblador se mantuvo durante años en el área de sistemas mucho tiempo después de que otros dominios de aplicación comenzaron a emplear lenguajes de mayor nivel. Se diseñaron muchos lenguajes de programación para sistemas, como CPL y BCPL, pero nunca se usaron en forma amplia. El lenguaje C (sección 10.2) cambió todo eso. Con el desarrollo de un entorno competitivo en UNIX escrito principalmente en C durante los primeros años de la década de 1970, se ha demostrado la eficacia de lenguajes de alto nivel en este entorno, así como en otros.

1.2.2 El papel de los lenguajes de programación

Inicialmente, los lenguajes se proyectaban para ejecutar programas con eficiencia. Las computadoras, que costaban millones de dólares, eran el recurso crítico, mientras que los programadores, que ganaban quizá 10 000 dólares anuales, constituían un costo menor. Cualquier lenguaje de alto nivel tenía que ser competitivo con el comportamiento de ejecución del lenguaje ensamblador codificado a mano. John Backus, el principal diseñador de FORTRAN para IBM a finales de los años cincuenta, afirmó una década más tarde [IBM 1966]:

> Francamente, no teníamos la menor idea cómo iba a resultar la cosa [el lenguaje FORTRAN y el compilador] con detalle. ... Salimos simplemente a optimizar el programa objeto, el tiempo de ejecución, porque la mayoría de la gente creía en esa época que en realidad no se podía hacer esa clase de cosa. Pensaban que los programas codificados por máquina serían tan terriblemente ineficientes que resultarían imprácticos para muchísimas aplicaciones.
>
> Un resultado en el que no pensábamos era este asunto de tener un sistema diseñado para ser completamente independiente de la máquina en la que el programa iba a ejecutarse en último término. Resultó ser una capacidad muy valiosa, pero sin duda que no la teníamos en mente.
>
> Nada había de organizado en nuestras actividades. Cada parte del programa fue escrita por una o dos personas que eran maestros absolutos de lo que hacían, con mínimas excepciones — y la cosa simplemente creció desordenadamente— ... [Cuando FORTRAN se distribuyó] tuvimos el problema de enfrentar el hecho de que estas 25 000 instrucciones no iban a estar todas correctas, y que iban a presentarse dificultades que se habrían de manifestar sólo después de mucho uso.

Para mediados de los años sesenta, cuando se expresó la cita precedente, y después del advenimiento de FORTRAN, COBOL, LISP Y ALGOL, Backus ya se había dado cuenta que la programación estaba cambiando. Las máquinas se estaban haciendo menos costosas, los costos de programación iban en aumento, existía la necesidad de trasladar programas de un sistema a otro y el mantenimiento del producto resultante estaba consumiendo una proporción mayor de los recursos de cómputo. En vez de compilar programas para trabajar con eficiencia en una computadora grande y costosa, la tarea de un lenguaje de alto nivel era facilitar el desarrollo de programas correctos para resolver problemas en alguna área de aplicación dada.

La tecnología de compiladores maduró en los años sesenta y setenta (capítulo 3) y la tecnología de lenguajes se centró en resolver problemas específicos en cuanto a dominio. El cómputo científico empleaba en general FORTRAN, las aplicaciones de negocios se escribían típicamente en COBOL, las aplicaciones militares se escribían en JOVIAL, las de inteligencia artificial se escribían en LISP y las aplicaciones militares incrustadas se tenían que escribir en Ada.

Al igual que los lenguajes naturales, los lenguajes de programación evolucionan o dejan de usarse. El ALGOL de 1960 ya no se usa, el uso de COBOL para aplicaciones de negocios está decayendo, y APL, PL/I y SNOBOL4, todos de los años sesenta, han desaparecido casi

por completo. Para el Pascal, de principios de los años setenta, ya pasó hace mucho su mejor momento, aunque muchas de sus construcciones continúan en Ada.

Los lenguajes más antiguos que todavía se usan han sufrido revisiones periódicas para reflejar las influencias cambiantes de otras áreas de la computación. FORTRAN ha sufrido varias revisiones de normalización, al igual que COBOL. Ada tiene una nueva norma de 1995; LISP ha sido actualizado con Scheme y más tarde con Common LISP. Los nuevos lenguajes como C++ y ML reflejan una combinación de experiencia adquirida en el diseño y uso de los mismos y de los cientos de otros lenguajes más antiguos. Algunas de estas influencias son:

1. *Capacidades de las computadoras.* Las computadoras han evolucionado, de las pequeñas, lentas y costosas máquinas de tubos de vacío de los años cincuenta, a las supercomputadoras y microcomputadoras de nuestros días. Al mismo tiempo, se han insertado capas de software de sistemas operativos entre el lenguaje de programación y el hardware subyacente de la computadora. Por ejemplo, partes de este libro se escribieron en una computadora que trabajaba alrededor de cinco veces más rápido, y sin embargo costó sólo una quinta parte del 1% del costo de la computadora que se usó para escribir la primera edición de este libro. Estos factores han influido tanto sobre la estructura como sobre el costo de utilizar las capacidades de los lenguajes de alto nivel.
2. *Aplicaciones.* El uso de computadoras se ha extendido rápidamente, de la concentración original en aplicaciones militares, científicas, de negocios e industriales de los años cincuenta, donde se podía justificar el costo, a los juegos de computadora, computadoras personales y aplicaciones en todas las áreas de actividad humana que se observan hoy día. Los requerimientos de estas nuevas áreas de aplicación afectan los diseños de nuevos lenguajes y las revisiones y ampliaciones de los más antiguos.
3. *Métodos de programación.* Los diseños de los lenguajes han evolucionado para reflejar nuestra cambiante comprensión de los buenos métodos para escribir programas grandes y complejos y para reflejar los cambios en el entorno en el cual se hace la programación.
4. *Métodos de implementación.* El desarrollo de mejores métodos de implementación ha influido en la selección de características que se habrán de incluir en los nuevos diseños.
5. *Estudios teóricos.* La investigación de las bases conceptuales del diseño e implementación de lenguajes, a través del uso de métodos matemáticos formales, ha profundizado nuestra comprensión de las fortalezas y debilidades de las características de los lenguajes y, por tanto, ha influido en la inclusión de estas características en los nuevos diseños de lenguajes.
6. *Estandarización.* La necesidad de lenguajes "estándar" que se pueden implementar con facilidad en una variedad de computadoras y que permiten transportar programas de una computadora a otra ha constituido una fuerte influencia conservadora sobre la evolución de los diseños de lenguajes.

Como ilustración, la tabla 1.1 presenta una lista breve de algunos de los lenguajes e influencias tecnológicas que fueron importantes durante cada periodo de cinco años, de 1950 a 1995. Muchos de estos temas se vuelven a tratar en capítulos posteriores. Desde luego, en

Años	Influencias y nueva tecnología
1951-55	**Hardware:** Computadoras de tubos de vacío; memorias lineales de retardo de mercurio **Métodos:** Lenguajes ensambladores; conceptos fundamentales: subprogramas, estructuras de datos **Lenguajes:** Uso experimental de compiladores de expresiones
1956-60	**Hardware:** Almacenamiento en cintas magnéticas; memorias de núcleo; circuitos de transistores **Métodos:** Tecnología inicial de compiladores; gramáticas BNF; optimización de códigos; intérpretes; métodos de almacenamiento dinámico y procesamiento de listas **Lenguajes:** FORTRAN, ALGOL 58, ALGOL 60, COBOL, LISP
1961-65	**Hardware:** Familias de arquitecturas compatibles; almacenamiento en discos magnéticos **Métodos:** Sistemas operativos de multiprogramación; compiladores dirigidos por sintaxis **Lenguajes:** COBOL 61, ALGOL 60 (revisado), SNOBOL, JOVIAL, notación APL
1966-70	**Hardware:** Tamaño y velocidad crecientes y costo decreciente; minicomputadoras; microprogramación; circuitos integrados **Métodos:** Sistemas de compartición de tiempo e interactivos; compiladores de optimización; sistemas de escritura de traductores **Lenguajes:** APL, FORTRAN 66, COBOL 65, ALGOL 68, SNOBOL 4, BASIC, PL/I, SIMULA 67, ALGOL-W
1971-75	**Hardware:** Microcomputadoras; edad de las minicomputadoras; sistemas pequeños de almacenamiento en masa; decadencia de las memorias de núcleo y ascenso de las memorias de semiconductores **Métodos:** Verificación de programas; programación estructurada; crecimiento inicial de la ingeniería de software como disciplina de estudio **Lenguajes:** Pascal, COBOL 74, PL/I (estándar), C, Scheme, Prolog
1976-80	**Hardware:** Microcomputadoras de calidad comercial; sistemas grandes de almacenamiento en masa; computación distribuida **Métodos:** Abstracción de datos, semántica formal; técnicas de programación concurrente, incrustada y en tiempo real **Lenguajes:** Smalltalk, Ada, FORTRAN 77, ML
1981-85	**Hardware:** Computadoras personales; primeras estaciones de trabajo; videojuegos, redes de área local; Arpanet **Métodos:** Programación orientada a objetos; entornos interactivos; editores dirigidos por sintaxis **Lenguajes:** Turbo Pascal, Smalltalk-80, crecimiento de Prolog, Ada 83, Postscript
1986-90	**Hardware:** Edad de la microcomputadora; ascenso de la estación de trabajo de ingeniería; arquitecturas RISC; redes globales; Internet **Métodos:** Computación de cliente/servidor **Lenguajes:** FORTRAN 90, C++, SML (Standard ML)
1991-95	**Hardware:** Estaciones de trabajo y microcomputadoras muy rápidas y de bajo costo; arquitecturas masivamente paralelas; voz, video, fax, multimedia **Métodos:** Sistemas abiertos; marcos de entorno; National Information Infrastructure ("supercarretera de la información") **Lenguajes:** Ada 95, lenguajes de proceso (TCL, PERL)

Tabla 1.1. Algunas influencias importantes sobre los lenguajes de programación.

esta tabla están ausentes los cientos de lenguajes e influencias que han desempeñado un papel menor, aunque importante de cualquier forma, en esta historia.

1.3 ¿QUÉ ES LO QUE CONSTITUYE UN BUEN LENGUAJE?

Todavía es necesario perfeccionar los mecanismos para el diseño de lenguajes de alto nivel. Todos los lenguajes de este libro tienen defectos, pero cada uno también es exitoso en comparación con los muchos cientos de otros lenguajes que han sido diseñados, implementados y utilizados por un tiempo, para luego caer en desuso.

Algunas de las razones del éxito o fracaso de un lenguaje pueden ser externas al lenguaje mismo. Por ejemplo, el uso de COBOL o Ada en Estados Unidos se impuso en ciertas áreas de programación por mandato del gobierno. De manera similar, parte de las razones del éxito del FORTRAN pueden atribuirse al fuerte apoyo de diversos fabricantes de computadoras que han invertido grandes esfuerzos para proveer implementaciones refinadas y una documentación extensa para estos lenguajes. Parte del éxito de SNOBOL4 durante los años setenta se puede atribuir a un texto excelente que describe el lenguaje [GRISWOLD 1975]. Pascal y LISP se han visto beneficiados por su uso como objetos de estudio teórico por parte de estudiantes de diseño de lenguajes, así como por su empleo práctico efectivo.

1.3.1 Atributos de un buen lenguaje

A pesar de la gran importancia de algunas de estas influencias externas, es el programador quien determina en último término, aunque a veces de manera indirecta, cuáles lenguajes viven o mueren. Se podrían sugerir múltiples razones para explicar por qué los programadores prefieren un lenguaje respecto a otro. Consideremos algunas de ellas.

1. *Claridad, sencillez y unidad.* Un lenguaje de programación proporciona a la vez un marco conceptual para pensar acerca de los algoritmos y un medio de expresar esos algoritmos. El lenguaje debe constituir una ayuda para el programador mucho antes de la etapa misma de codificación. Debe proveer un conjunto claro, sencillo y unificado de conceptos que se puedan usar como primitivas en el desarrollo de algoritmos. Para ello, es deseable contar con un número mínimo de conceptos distintos cuyas reglas de combinación sean tan sencillas y regulares como sea posible. Llamamos a este atributo *integridad conceptual.*

   La sintaxis de un lenguaje afecta la facilidad con la que un programa se puede escribir, poner a prueba, y más tarde entender y modificar. La *legibilidad* de los programas en un lenguaje es aquí una cuestión medular. Una sintaxis que es particularmente tersa o hermética suele facilitar la escritura de un programa (para el programador con experiencia) pero dificulta su lectura cuando es necesario modificarlo más tarde. Los programas en APL suelen ser tan herméticos que sus propios diseñadores no pueden descifrarlos pocos meses después de haberlos completado. Muchos lenguajes contienen construcciones sintácticas que favorecen una lectura errónea al hacer que dos enunciados

casi idénticos signifiquen en efecto cosas radicalmente distintas. Por ejemplo, la presencia de un carácter en blanco, que es un operador, en un enunciado en SNOBOL4 puede alterar por completo su significado. Un lenguaje debe tener la propiedad de que las construcciones que *signifiquen* cosas distintas se vean diferentes; es decir, las diferencias semánticas deberán reflejarse en la sintaxis del lenguaje.

2. *Ortogonalidad.* El término *ortogonalidad* se refiere al atributo de ser capaz de combinar varias características de un lenguaje en todas las combinaciones posibles, de manera que todas ellas tengan significado. Por ejemplo, supóngase que un lenguaje dispone de una expresión que puede producir un valor, y también dispone de un enunciado condicional que evalúa una expresión para obtener un valor de verdadero o falso. Estas dos características del lenguaje, la expresión y el enunciado condicional, son ortogonales si se puede usar (y evaluar) *cualquier* expresión dentro del enunciado condicional.

   Cuando las características de un lenguaje son ortogonales, entonces es más fácil aprender el lenguaje y escribir los programas porque hay menos excepciones y casos especiales que recordar. El aspecto negativo de la ortogonalidad es que un programa suele compilar sin errores a pesar de contener una combinación de características que son lógicamente incoherentes o cuya ejecución es en extremo ineficiente. A causa de estas cualidades en oposición, la ortogonalidad como atributo de un diseño de lenguaje es todavía motivo de controversia, pues a ciertas personas les gusta y a otras no.

3. *Naturalidad para la aplicación.* Un lenguaje necesita una sintaxis que, al usarse correctamente, permita que la estructura del programa refleje la estructura lógica subyacente del algoritmo. Idealmente, deberá ser posible traducir directamente un diseño de programa de este tipo a enunciados de programa adecuados que reflejen la estructura del algoritmo. Los algoritmos secuenciales, algoritmos concurrentes, algoritmos lógicos, etc., todos ellos tienen estructuras naturales diferentes que están representadas por programas en esos lenguajes.

   El lenguaje deberá suministrar estructuras de datos, operaciones, estructuras de control y una sintaxis natural apropiada para el problema que se va a resolver. Una de las razones principales de la proliferación de lenguajes es precisamente esta necesidad de naturalidad. Un lenguaje particularmente adecuado para una cierta clase de aplicaciones puede simplificar grandemente la creación de programas individuales en esa área. Prolog, con su predisposición hacia propiedades deductivas, y C++, para diseño orientado a objetos, son dos lenguajes de la parte II con una inclinación obvia hacia clases particulares de aplicaciones.

4. *Apoyo para la abstracción.* Incluso con el lenguaje de programación más natural para una aplicación, siempre queda una brecha considerable entre las estructuras de datos y operaciones abstractas que caracterizan la solución de un problema y las estructuras de datos primitivos y operaciones particulares integradas en un lenguaje. Por ejemplo, C puede ser un lenguaje apropiado para construir un programa para organizar los horarios de clases en una universidad, pero las estructuras de datos abstractos de "estudiante", "sección de clase", "instructor", "salón de clases" y las operaciones abstractas de "asignar un estudiante a una sección de clase", "organizar una sección de

clase en un salón de clases," etc., que son naturales para la aplicación no son suministradas directamente por C.

Una parte considerable de la tarea del programador es proyectar las abstracciones adecuadas para la solución del problema y luego implementar esas abstracciones empleando las capacidades más primitivas que provee el lenguaje de programación mismo. Idealmente, el lenguaje debe permitir la definición y el mantenimiento de las estructuras de datos, de los tipos de datos y de las operaciones como abstracciones autosuficientes. El programador puede emplear todo esto en otras partes del programa conociendo sólo sus propiedades abstractas, sin preocuparse por los detalles de su implementación. Tanto Ada como C++ se desarrollaron debido precisamente a estos defectos de los lenguajes más antiguos Pascal y C, respectivamente.

5. *Facilidad para verificar programas.* La confiabilidad de los programas escritos en un lenguaje es siempre una preocupación medular. Existen muchas técnicas para verificar que un programa ejecuta correctamente la función requerida. Se puede *probar que un programa es correcto* a través de un método formal de verificación (véase la sección 9.4), se puede probar informalmente que es correcto por *verificación de escritorio* (leer y verificar visualmente el texto del programa), se puede *poner a prueba* ejecutándolo con los datos de entrada de prueba y comparando los resultados de salida con las especificaciones, etc. Para programas grandes se suele emplear alguna combinación de todos estos métodos. El uso de un programa que dificulta la verificación de programas puede ser mucho más problemático que el de uno que apoya y simplifica la verificación, no obstante que el primero pueda proporcionar muchas capacidades que superficialmente parecen hacer más fácil la programación. La sencillez de la estructura semántica y sintáctica es un aspecto primordial que tiende a simplificar la verificación de programas.

6. *Entorno de programación.* La estructura técnica de un lenguaje de programación es sólo uno de los aspectos que afectan su utilidad. La presencia de un *entorno de programación* adecuado puede facilitar el trabajo con un lenguaje técnicamente débil en comparación con un lenguaje más fuerte con poco apoyo externo. Se podría incluir una larga lista de factores como parte del entorno de programación. La disponibilidad de una implementación confiable, eficiente y bien documentada del lenguaje debe encabezar la lista. Los editores especiales y paquetes de prueba hechos a la medida para el lenguaje pueden acelerar mucho la creación y puesta a prueba de programas. Los recursos para el mantenimiento y modificación de múltiples versiones de un programa pueden simplificar mucho el trabajo con programas grandes. De los lenguajes que se estudian en este libro, sólo Smalltalk fue proyectado específicamente alrededor de un entorno de programación compuesto de ventanas, menús, uso del ratón y un conjunto de herramientas para operar sobre programas escritos en Smalltalk.

7. *Portabilidad de programas.* Un criterio importante para muchos proyectos de programación es el de la *portabilidad* de los programas resultantes de la computadora en la cual se desarrollaron hacia otros sistemas de computadoras. Un lenguaje que está ampliamente disponible y cuya definición es independiente de las características de una máquina particular constituye una base útil para la producción de programas trans-

portables. Ada, FORTRAN, C y Pascal tienen todos ellos definiciones estandarizadas que permiten la implementación de aplicaciones transportables. Otros, como ML, provienen de una implementación de fuente única que permite al diseñador de lenguajes cierto control sobre las características transportables del lenguaje.

8. *Costo de uso.* Se ha dejado para el final el engañoso criterio del costo. Es indudable que el costo es un elemento importante en la evaluación de cualquier lenguaje de programación, pero son factibles diferentes medidas del mismo:

(a) Costo de ejecución del programa. En los primeros días de la computación, las cuestiones de costo se referían casi exclusivamente a la ejecución de los programas. Era importante el diseño de compiladores que optiman, la asignación eficiente de registros y el diseño de mecanismos eficientes de apoyo al tiempo de ejecución. El costo de ejecución del programa, aunque siempre ha tenido cierta importancia en el diseño de lenguajes, es de importancia primordial para grandes programas de producción que se van a ejecutar con frecuencia. En la actualidad, sin embargo, para muchas aplicaciones la velocidad de ejecución no es la preocupación mayor. Con máquinas de escritorio que trabajan a varios millones de instrucciones por segundo y que están ociosas gran parte del tiempo, se puede tolerar un incremento de 10% o 20% del tiempo de ejecución si ello significa un mejor diagnóstico o un control más fácil por parte del usuario sobre el desarrollo y mantenimiento del programa.

(b) Costo de traducción de programas. Cuando un lenguaje como FORTRAN o C se utiliza en la enseñanza, la cuestión de una traducción (compilación) eficiente, más que la ejecución eficiente, puede ser de importancia capital. Típicamente, los programas estudiantiles se compilan muchas veces cuando se están depurando pero se ejecutan sólo unas pocas veces. En casos así, es importante contar con un compilador rápido y eficiente en vez de un compilador que produzca un código ejecutable optimizado.

(c) Costo de creación, prueba y uso de programas. Un tercer aspecto del costo de un lenguaje de programación queda ejemplificado por el lenguaje Smalltalk. Para una cierta clase de problemas se puede diseñar, codificar, probar, modificar y usar una solución con una inversión mínima en tiempo y energía del programador. Smalltalk es económico en cuanto a que se minimizan el tiempo y esfuerzo totales que se invierten en la solución de un problema en la computadora. La preocupación por esta clase de costo de conjunto en el uso de un lenguaje se ha vuelto tan importante en muchos casos como la inquietud por la ejecución y compilación eficiente de los programas.

(d) Costo de mantenimiento de los programas. Muchos estudios han demostrado que el costo más grande que interviene en cualquier programa que se utiliza a lo largo de un periodo de años no es el costo del diseño, codificación y prueba inicial del programa, sino los *costos totales del ciclo vital*, que incluyen los costos de desarrollo y el costo de mantenimiento del programa en tanto continúa en uso productivo. El mantenimiento incluye la reparación de errores, los que se descubren después de que se comienza a usar el programa, cambios que requiere el programa cuando se actualiza el hardware subyacente o el sistema operativo, y extensiones y mejoras al programa que se requieren para satisfacer nuevas necesidades. Un lenguaje que facilita la modificación, reparación

Época	Aplicación	Lenguajes principales	Otros lenguajes
Años sesenta	Negocios	COBOL	Ensamblador
	Científica	FORTRAN	ALGOL, BASIC, APL
	Sistemas	Ensamblador	JOVIAL, Forth
	IA	LISP	SNOBOL
Hoy	Negocios	COBOL, hoja de cálculo	C, PL/I, 4GL
	Científica	FORTRAN, C, C++	BASIC, Pascal
	Sistemas	C,C++	Pascal, Ada, BASIC, Modula
	IA	LISP, Prolog	
	Edición	TeX, Postscript, procesamiento de texto	
	Proceso	Shell de UNIX, TCL, PERL	Marvel
	Nuevos paradigmas	ML, Smalltalk	Eiffel

Tabla 1.2. Lenguajes para diversos dominios de aplicación.

y extensión frecuente del programa por parte de diferentes programadores durante un periodo de varios años puede ser, a la larga, mucho menos costoso de usar que cualquier otro.

1.3.2 Dominios de aplicación

El lenguaje apropiado que se use a menudo, depende del dominio de la aplicación que resuelve el problema. El lenguaje adecuado que conviene usar para diversos dominios de aplicación ha evolucionado a lo largo de los últimos 30 años. Los lenguajes que se estudian en este libro, más algunos otros, se resumen en la tabla 1.2.

Aplicaciones de los años sesenta

Durante la década de 1960, casi toda la programación se podía dividir en cuatro modelos básicos de programación: aplicaciones de procesamiento de negocios, cálculos científicos, programación de sistemas e inteligencia artificial.

De procesamiento de negocios. Casi siempre se trataba de grandes aplicaciones de procesamiento de datos proyectadas para ejecutarse en computadoras centrales. Incluían programas para introducción de pedidos, control de inventarios, administración de personal y nómina. Se caracterizaban por leer grandes cantidades de datos en unidades con varias cintas; leían en un conjunto más pequeño de transacciones recientes y escribían un nuevo conjunto de datos históricos. Para darse una idea del aspecto de todo esto, véase cualquier película de ciencia ficción de los años sesenta. Se acostumbraba mostrar una gran cantidad de cintas girando para ambientar la "computación moderna".

COBOL se desarrolló para estas aplicaciones. Los diseñadores de COBOL se esforzaron mucho para asegurar que esta clase de registros de procesamiento de datos se habrían de procesar de manera correcta. Las aplicaciones de negocios también incluyen la planeación de los mismos, análisis de riesgos y escenarios de "qué pasa si". En los sesenta, a menudo un programador de COBOL, requería varios meses para terminar una aplicación típica "qué pasa si".

Científicas. Estas aplicaciones se caracterizan por la solución de diversas ecuaciones matemáticas. Incluyen problemas de análisis numérico, solución de funciones diferenciales o integrales, y generación de estadísticas. Es en este campo donde la computadora tuvo su desarrollo inicial, para usarse durante la Segunda Guerra Mundial para generar tablas de balística. FORTRAN ha sido siempre el lenguaje dominante en esta esfera. Su sintaxis siempre se ha aproximado a las matemáticas y los científicos lo encuentran fácil de usar.

De sistemas. Para construir sistemas operativos y para utilizar compiladores no existía un lenguaje eficaz. Las aplicaciones de este tipo deben ser capaces de tener acceso a toda la funcionalidad y los recursos del hardware subyacente. El lenguaje ensamblador solía ser la opción para lograr eficiencia. El JOVIAL, una variante del ALGOL, se empleó en ciertos proyectos del Departamento de Defensa de Estados Unidos, y hacia finales de los años sesenta se usaban lenguajes como el PL/I para esta aplicación.

Un dominio de aplicación relacionado es el control de procesos, la regulación o el manejo de maquinaria. A causa del costo y dimensiones de las computadoras durante esta época, casi todas las aplicaciones de control eran grandes, por ejemplo para gobernar una planta generadora de electricidad o una línea automática de ensamble. Se desarrollaron lenguajes como Forth para tratar con este dominio de aplicación, aunque se solía usar lenguaje ensamblador.

De IA. La inteligencia artificial era un área de investigación relativamente nueva, y LISP era el lenguaje dominante para aplicaciones de IA. Estos programas se caracterizan por algoritmos que buscan a través de grandes espacios de datos. Por ejemplo, para jugar ajedrez, la computadora genera muchos movimientos potenciales y luego busca el mejor de ellos dentro del tiempo de que dispone para decidir lo que tiene que hacer a continuación.

Aplicaciones de los años noventa

Aunque Ada se desarrolló para eliminar gran parte de la duplicación entre lenguajes en competencia, la situación actual es probablemente más compleja de lo que era durante la década de 1960. Tenemos más dominios de aplicación donde los lenguajes de programación se adaptan especialmente bien, con múltiples opciones para cada dominio de aplicación.

De procesamiento de negocios. En este dominio, COBOL es todavía el lenguaje dominante para aplicaciones de procesamiento de datos, aunque a veces se utilizan C y PL/I. Sin embargo, el escenario de "qué pasa si" ha cambiado por completo. En nuestros días la hoja de cálculo ha reformado este dominio de aplicación. Mientras que en otra época le tomaba al programador varios meses elaborar un programa típico de planeación de negocios, en la actualidad un analista puede "cocinar" muchas hojas de cálculo en unas pocas horas.

Los lenguajes de cuarta generación (4GL) se han apoderado también de parte de este mercado. Los 4GL son lenguajes adaptados para dominios específicos de aplicaciones de negocios y suministran típicamente una interfaz de programador con base en ventanas, fácil acceso a registros de bases de datos y capacidades especiales para generar formas de entrada de "llenado de espacios en blanco" y elegantes informes de salida. A veces estos "compiladores" de 4GL generan programas en COBOL como salida.

Científicas. También el FORTRAN anda todavía por ahí. Aunque el FORTRAN 77 está enfrentando el reto de lenguajes como C, el FORTRAN 90 es una actualización reciente del estándar del lenguaje, la cual incorpora muchas características que se encuentran en Ada y otros lenguajes.

De sistemas. El C, desarrollado hacia finales de los años sesenta, y su nueva variante C++, imperan en este dominio de aplicación. El C proporciona una ejecución muy eficiente y permite al programador tener pleno acceso al sistema operativo y al hardware subyacente. También se usan otros lenguajes como Modula, Pascal, y variantes modernas del BASIC. Aunque destinado a esta área, el Ada nunca ha alcanzado su objetivo de convertirse en un lenguaje importante en este dominio. La programación en lenguaje ensamblador se ha vuelto anacrónica.

Con el advenimiento de microprocesadores baratos que gobiernan automóviles, hornos de microondas, juegos de video y relojes digitales, ha aumentado la necesidad de contar con lenguajes para tiempo real. C, Ada y Pascal se suelen usar para este tipo de procesamiento en tiempo real.

De IA. Todavía se utiliza LISP, aunque las versiones modernas como Scheme y Common LISP han reemplazado al LISP 1.5 del MIT de principios de los años sesenta. Prolog ha desarrollado un buen número de seguidores. Ambos lenguajes son muy aptos para aplicaciones de "búsqueda".

Edición. La edición representa un dominio de aplicación relativamente nuevo para los lenguajes. Los sistemas de procesamiento de texto tienen su propia sintaxis para mandatos de entrada y archivos de salida. La composición de este libro se hizo utilizando el sistema de procesamiento de texto TEX y, a falta de un mejor término, los capítulos se "compilaron" conforme se escribían para introducir referencias de figuras y tablas, colocar figuras y componer párrafos.

El traductor TEX produce un programa en el lenguaje Postscript de descripción de páginas. Aunque Postscript es ordinariamente la salida de un procesador, tiene sintaxis y semántica y se puede compilar por medio de un procesador adecuado. Éste suele ser la impresora láser que se utiliza para imprimir el documento. Sabemos de individuos que insisten en programar directamente en Postscript, pero esto parece ser tan tonto en la actualidad como lo era la programación en lenguaje ensamblador en los años sesenta.

De proceso. Durante los años sesenta el programador era el agente activo en cuanto al uso de una computadora. Para llevar a cabo una tarea, el programador escribía un comando apropiado que la computadora ejecutaba luego. Sin embargo, en la actualidad solemos usar un programa para controlar otro, por ejemplo, para hacer el respaldo de archivos todas las noches, para sincronizar el tiempo cada hora, para enviar una respuesta automática al correo electrónico que llega cuando estamos de vacaciones, para poner automáticamente a prueba un programa

cada vez que compila satisfactoriamente, etc. Llamamos a estas actividades *procesos*, y existe un interés considerable en el desarrollo de lenguajes donde estos procesos se puedan especificar y luego traducir para ejecutarlos en forma automática.

Dentro de UNIX, el lenguaje de comandos de usuario se conoce como *shell* (concha) y a los programas se les llama *guiones de shell*. Estos guiones se pueden invocar siempre que ocurren ciertas condiciones habilitadoras. Han aparecido varios lenguajes de guiones; tanto el TCL como el PERL se usan para fines parecidos. El archivo "*.BAT" de MS-DOS es una forma sencilla de guión.

Nuevos paradigmas. Siempre hay nuevos modelos de aplicación en estudio. El ML se ha utilizado en la investigación de lenguajes de programación para investigar la teoría de tipos. Aunque no es un lenguaje muy extendido en la industria, su popularidad va en aumento. Smalltalk es otro lenguaje importante. Aunque el uso comercial de Smalltalk no es muy grande, ha tenido un profundo efecto sobre el diseño de lenguajes. Muchas de las características orientadas a objetos de C++ y Ada tuvieron su origen en Smalltalk.

Los lenguajes para diversos dominios de aplicación son una fuente continua de nueva investigación y desarrollo. En la medida en que el conocimiento de las técnicas de compilación mejora, y conforme evoluciona el conocimiento de cómo construir sistemas complejos; constantemente se encuentran nuevos dominios de aplicación y se requieren lenguajes que satisfagan las necesidades de los mismos.

1.3.3 Estandarización de lenguajes

¿Qué es lo que describe un lenguaje de programación? Considérese el siguiente código de C:

```
int i; i = (1 && 2) + 3;
```

¿Es esto C válido y cuál es el valor de *i*? ¿Cómo respondería usted estas preguntas?[1] Son tres los enfoques que se usan con más frecuencia:

1. Lea la definición en el manual de referencia del lenguaje y decida qué significa el enunciado.
2. Escriba un programa en su sistema local de computadora y vea lo que ocurre.
3. Lea la definición en el estándar del lenguaje.

La opción 2 es probablemente la más común. Simplemente siéntese y escriba un programa de dos o tres líneas que ponga a prueba esta condición. Por tanto, el concepto de lenguaje de programación está estrechamente ligado con la implementación particular del lenguaje en su sistema de computadora. Para alguien más "académico", también se puede revisar un manual de referencia del lenguaje, que publica típicamente el proveedor del compilador local de C. Puesto que pocos tienen acceso al estándar del lenguaje, rara vez se utiliza la opción 3.

Las opciones 1 y 2 significan que el concepto de lenguaje de programación está ligado a una implementación particular. Pero, ¿es correcta esa implementación? ¿Qué ocurre si usted desea trasladar su programa de 50,000 líneas en C a otra computadora que tiene un compilador

[1] Adviértase que las respuestas son sí y 4, respectivamente, para quien tenga curiosidad.

de un proveedor distinto? ¿Continuará compilando correctamente el programa y producirá los mismos resultados cuando se ejecute? Si no es así, ¿por qué no? Con frecuencia, el diseño incluye ciertos detalles complicados y un proveedor puede tener una interpretación distinta de la de otro, lo cual produce un comportamiento de ejecución ligeramente diferente.

Por otra parte, un proveedor puede decidir que una nueva capacidad incorporada al lenguaje puede hacerlo más útil. ¿Es esto "legal"? Por ejemplo, si se amplía C para agregarle una nueva declaración dinámica de arreglos, se le puede llamar todavía C al lenguaje? En tal caso, los programas que utilicen esta nueva capacidad en el compilador local no podrán compilar si se trasladan a otro sistema.

Para enfrentar estas preocupaciones, casi todos los lenguajes tienen definiciones estándar. Todas las implementaciones deben apegarse a esta "norma". Los estándares son en general de dos clases:

1. *Estándares patentados.* Son las definiciones elaboradas por la compañía que desarrolló el lenguaje y que es su propietaria. En casi todos los casos, los estándares patentados no se aplican a los lenguajes que se han popularizado y se usan ampliamente. Pronto aparecen variaciones en las implementaciones, con muchas mejoras e incompatibilidades.
2. *Estándares por consenso.* Se trata de documentos elaborados por organizaciones con base en un acuerdo entre los participantes pertinentes. Los estándares por consenso, o simplemente *estándares*, constituyen el método principal para asegurar la uniformidad entre varias implementaciones de un lenguaje.

Cada país cuenta típicamente con una organización a la que se ha asignado el papel de desarrollar normas. En Estados Unidos, es el American National Standards Institute (Instituto Nacional Estadounidense para Normas), o ANSI, con el papel relativo a normas de lenguajes de programación asignado al comité X3 de la Computer Business Equipment Manufacturers Association (Asociación de Fabricantes de Equipo de Cómputo para Negocios), o CBEMA. El Institute of Electrical and Electronic Engineers (Instituto de Ingenieros Eléctricos y Electrónicos), o IEEE, también puede desarrollar este tipo de normas. En el Reino Unido, el papel en cuanto a normas es asumido por el British Standards Institute (Instituto Británico para Normas), BSI. La International Standards Organization (ISO; Organización Internacional para Normas), cuya oficina principal está en Ginebra, Suiza, elabora las normas internacionales.

El desarrollo de normas sigue un proceso similar en todas estas organizaciones. En un momento dado, un grupo decide que un lenguaje requiere una definición estándar. El organismo normativo organiza un grupo de trabajo de voluntarios para desarrollar esa norma. Cuando el grupo de trabajo llega a un acuerdo sobre su norma, se somete a votación por parte de un bloque más grande de individuos interesados. Los desacuerdos se resuelven y se produce el estándar del lenguaje.

Aunque suena bien en teoría, la aplicación de la elaboración de estándares es en parte técnica y en parte política. Por ejemplo, los proveedores de compiladores tienen un fuerte interés financiero en el proceso de los estándares. Después de todo, desean que el estándar se parezca a su compilador actual para no tener que hacer cambios en su propia implementación. No sólo son costosos estos cambios, sino que los usuarios del compilador que emplean las características que han cambiado ahora tienen programas que no satisfacen la norma. Esto produce clientes descontentos.

Por consiguiente, como ya se ha afirmado, la elaboración de estándares es un proceso de consenso. No todo el mundo se sale con la suya, pero uno espera que el lenguaje resultante sea aceptable para todos. Considérese el ejemplo siguiente. Durante las deliberaciones para el estándar de FORTRAN de 1977, se acordó de manera general que las cadenas y subcadenas eran características deseables, puesto que casi todas las implementaciones de FORTRAN ya las tenían. Pero había varias implementaciones factibles de subcadenas: Si M = "abcdefg", entonces la subcadena "bcde" podría ser la cadena desde el segundo hasta el quinto carácter de M (M[2:5]) o la cadena que empieza en la posición 2 y se extiende a lo largo de 4 caracteres (M[2:4]). También se podría escribir como (M[3:6]) contando los caracteres desde la derecha. Puesto que no se podía llegar a un consenso, el "consenso" fue simplemente dejar esto fuera del estándar. Si bien no satisfacía la mayor parte de las metas para un lenguaje según se han expresado en este capítulo, fue la solución conveniente la que se adoptó. Por esta razón, los estándares son documentos útiles, pero la definición del lenguaje puede verse matizada por la política del momento.

Para utilizar los estándares en forma eficaz, es necesario ocuparse de tres cuestiones:

1. *Oportunidad.* ¿Cuándo estandarizar un lenguaje?
2. *Conformidad.* ¿Qué significa que un programa se adhiere a un estándar y que un compilador compila un estándar?
3. *Obsolescencia.* ¿Cuándo envejece un estándar y cómo se modifica?

A continuación se considera cada una de las preguntas.

Oportunidad. Una cuestión importante es *cuándo* se debe estandarizar un lenguaje. El FORTRAN se estandarizó inicialmente en 1966 cuando había muchas versiones incompatibles de "FORTRAN". Esto condujo a muchos problemas puesto que cada implementación era distinta de las otras. En el otro extremo, el Ada se estandarizó por primera vez en 1983 antes de que hubiera alguna implementación; por tanto, cuando se elaboró el estándar no estaba claro si el lenguaje funcionaría siquiera. Los primeros compiladores eficaces de Ada no aparecieron sino hasta 1987 o 1988, y estas primeras implementaciones identificaron varias idiosincrasias. Lo deseable sería estandarizar un lenguaje lo suficientemente pronto para que exista suficiente experiencia en el uso del lenguaje, pero no demasiado tarde, para no alentar muchas implementaciones incompatibles.

De los lenguajes de este libro, el FORTRAN se estandarizó tardíamente, cuando existían ya muchas variaciones incompatibles. El Ada se estandarizó muy pronto, antes que existieran implementaciones; C y Pascal se estandarizaron cuando su uso iba en aumento y antes que hubiera demasiadas versiones incompatibles.

De los otros lenguajes de este libro (LISP, Smalltalk, C++, ML y Prolog), ML existe en general como una sola implementación (Standard ML o SML) que todo el mundo usa, y casi todas las versiones de Smalltalk, C++ y Prolog son bastante semejantes, aunque hay variantes de C que agregan objetos que son similares a (pero diferentes de) C++. LISP es probablemente el que más ha sufrido por ser un lenguaje ampliamente utilizado sin una referencia estándar. Existen dialectos de LISP (Scheme, Common LISP, IBCL) y son similares, aunque incompatibles.

Conformidad. Si existe un estándar para un lenguaje, se suele hablar de *conformidad* con respecto a ese estándar. Un programa es *conforme* si sólo utiliza características definidas en el estándar. Un *compilador conformable* es uno que, cuando se le da un programa conforme, produce un programa ejecutable que genera la salida correcta.

Obsérvese que esto nada dice acerca de extensiones del estándar. Si un compilador agrega características adicionales, entonces cualquier programa que las utilice es no conformado, y el estándar no dice nada acerca de cuáles deberán ser los resultados del cómputo. En general, los estándares sólo se ocupan de programas conformes. Por esta razón, casi todos los compiladores tienen características de las que no se ocupa el estándar. Esto significa que se debe tener cuidado al utilizar la implementación local como autoridad última en cuanto al significado de una característica dada en un lenguaje. (Por ejemplo, véase el programa en Pascal *anomalía* en la figura 11.3.)

Obsolescencia. Conforme nuestro conocimiento y experiencia de la programación evolucionan, las nuevas arquitecturas de computadora requieren nuevas características de los lenguajes. Una vez que se estandariza un lenguaje, parece "arcaico" unos cuantos años después. El estándar original del FORTRAN 66 parece anticuado sin tipos, estructuras anidadas de control, encapsulamiento, estructura de bloques y las otras numerosas características de lenguajes más modernos.

El proceso de estandarización ya toma parte de esto en cuenta. Los estándares se tienen que revisar cada cinco años y ya sea renovarse o descartarse. El ciclo de cinco años suele alargarse un poco, pero el proceso es en gran medida eficaz. El FORTRAN fue estandarizado por primera vez en 1966, se revisó en 1978 (aunque se le llamó FORTRAN 77, a pesar de que no se alcanzó la fecha propuesta de conclusión de 1977 por unos cuantos meses) y una vez más en 1990. El Ada se estandarizó en 1983 y de nuevo en 1995.

Un problema en la actualización de un estándar es qué hacer con la colección existente de programas escritos para el estándar antiguo. Las compañías han invertido recursos significativos en su software, y la reescritura de todo este código para una nueva versión de un lenguaje es bastante costosa. Por esta razón, casi todos los estándares requieren compatibilidad hacia atrás; el nuevo estándar debe incluir versiones más antiguas del lenguaje.

Esto presenta problemas. Uno de ellos es que el lenguaje se puede volver difícil de manejar a causa de numerosas construcciones obsoletas. Algo más dañino es que algunas de estas construcciones pueden perjudicar el buen diseño de programas. El enunciado EQUIVALENCE (EQUIVALENCIA) del FORTRAN es un elemento de este tipo. Si A es un número real e I es un entero, entonces:

```
EQUIVALENCE (A,I)
A=A+1
I=I+1
```

asigna A e I a la misma ubicación de almacenamiento. La asignación a A accede a esta ubicación como número real y le suma 1. La asignación a I accede a esta ubicación suponiendo que es un entero y le suma 1. Puesto que la representación de enteros y reales es diferente en casi todas las computadoras, los resultados son en este caso muy poco predecibles. Dejar esta característica en el lenguaje no es una buena idea.

Recientemente se han desarrollado los conceptos de obsolescencia y características desaprobadas. Una característica es *obsolescente* si es candidata a ser descartada en la próxima versión del estándar. Esto advierte a los usuarios que la característica todavía está disponible, pero va a ser desechada en los próximos 5 o 10 años. Esto proporciona una larga advertencia para que se reescriba el código que utilice esa característica. Una característica *desaprobada* se puede volver obsolescente en el próximo estándar, por lo cual puede ser descartada después de dos revisiones. Esto proporciona una advertencia más prolongada, de 10 a 20 años. Los programas nuevos no deberán utilizar ninguna de ambas clases de características.

Puesto que se permite que los compiladores conformables a estándares tengan extensiones del lenguaje, siempre y cuando compilen correctamente los programas conformables al estándar, casi todos los compiladores tienen adiciones que el proveedor considera útiles y que van a incrementar la participación de mercado para ese producto. Esto permite que la innovación continúe y el lenguaje evolucione. Desde luego, dentro de la comunidad académica a casi todo el profesorado no le importan estos estándares y desarrolla sus propios productos, que amplían y modifican los lenguajes según se juzga conveniente. Esto provee una atmósfera fértil donde se ponen a prueba nuevas ideas en cuanto a lenguajes y algunas de las mejores se convierten en lenguajes y compiladores comerciales.

1.4 EFECTOS DE LOS ENTORNOS SOBRE LOS LENGUAJES

El desarrollo de un lenguaje de programación no tiene lugar en un vacío. El equipo (hardware) que apoya a un lenguaje tiene un gran impacto sobre el diseño del mismo. El lenguaje, como medio para resolver un problema, es parte de la tecnología global que se emplea. El entorno externo que apoya la ejecución de un programa se conoce como su *entorno operativo* u *objetivo*. El entorno en el cual un programa se diseña, se codifica, se pone a prueba y se depura, o *entorno anfitrión*, puede ser diferente del entorno operativo en el cual se usa el programa en último término.

Cuatro clases generales de entornos objetivo cubren casi todas las aplicaciones de programación: de procesamiento por lotes, interactivo, de sistema incrustado y de programación (como caso especial de entorno interactivo). Cada uno plantea distintos requerimientos sobre los lenguajes adaptados a esos entornos.

1.4.1 Entornos de procesamiento por lotes

El primero y más simple entorno operativo se compone sólo de archivos externos de datos (sección 4.3.12). Un programa toma un cierto conjunto de archivos de datos como entrada, procesa los datos y produce un conjunto de archivos de datos de salida; por ejemplo, un programa de nómina procesa dos archivos de entrada que contienen registros maestros de nómina y tiempos de periodos de paga semanales, y produce dos archivos de salida que contienen registros maestros actualizados y cheques de sueldo. Este entorno operativo se designa como de *procesamiento por lotes* porque los datos de entrada se reúnen en "lotes" de archivos y son procesados en lotes por el programa. Los lenguajes como FORTRAN, C y Pascal se proyectaron

inicialmente para entornos de procesamiento por lotes, aunque ahora se pueden usar en un entorno interactivo o en uno de sistema incrustado.

Efectos sobre el diseño de lenguajes. La influencia del entorno se aprecia en cuatro áreas principales: características de entrada/salida, características de manejo de errores y excepciones, recursos de regulación del tiempo y estructura de programas.

En un lenguaje proyectado para procesamiento por lotes, por lo común los archivos constituyen la base para casi toda la estructura de entrada/salida. Aunque un archivo se puede usar para entrada/salida interactiva a una terminal, en estos lenguajes no se encaran las necesidades especiales de la E/S interactiva. Por ejemplo, los archivos se guardan ordinariamente como registros de longitud fija, pero en una terminal el programa necesitaría leer cada carácter conforme se introduce en el teclado. La estructura de entrada/salida tampoco se ocupa típicamente de la necesidad de acceso a dispositivos especiales de E/S que se encuentran en sistemas incrustados.

En un entorno de procesamiento por lotes, un error que termine la ejecución del programa es aceptable aunque costoso, debido a que a menudo se repite toda la ejecución después de corregir el error. Además, en este entorno, no es posible la ayuda externa por parte del usuario para manejar o corregir el error de inmediato. Así pues, las facilidades del lenguaje para manejo de errores y excepciones enfatizan el manejo de éstos en el programa para que éste se pueda recuperar de casi todos los errores y continúe con el procesamiento sin suspender su ejecución.

Una tercera característica que distingue a un entorno de procesamiento por lotes es la carencia de restricciones de regulación de tiempo en un programa. Ordinariamente, el lenguaje no proporciona recursos para monitorear o afectar directamente la velocidad de ejecución del programa. Por ejemplo, el sistema operativo puede permitir que el programa se ejecute por un tiempo, luego intercambiarlo a almacenamiento secundario por un periodo indefinido, y más tarde traerlo de nuevo a la memoria central para continuar. El tiempo preciso que se requiere para completar la ejecución del programa no es algo que preocupe mucho (dentro de ciertos límites amplios). Más aún, la implementación de las características de los lenguajes suele reflejar la falta de restricciones de regulación del tiempo.

La estructura de programa representativa de este entorno consiste en un programa principal y un conjunto de subprogramas. Esta estructura típica de programa está presente en FORTRAN, C y Pascal. Los subprogramas, en su totalidad, son procesados por el compilador y éste interactúa rara vez con el programador durante el proceso de compilación.

1.4.2 Entornos interactivos

En un entorno interactivo, el más común en la actualidad en computadoras personales y estaciones de trabajo, un programa interactúa durante su ejecución directamente con un usuario en una consola de visualización, enviando alternativamente salidas hacia ésta y recibiendo entradas desde el teclado o ratón. Son ejemplos los sistemas de procesamiento de texto, hojas de cálculo, juegos de video, sistemas de gestión de bases de datos y sistemas de instrucción asistida por computadora. Todos estos ejemplos son todos herramientas con las cuales el lector puede estar familiarizado. En la sección 9.6 se analiza el diseño de sistemas que amplía

estos entornos interactivos a sistemas de procesamiento de transacciones (por ejemplo, un sistema de reservación en líneas aéreas) y redes de computadoras de cliente/servidor.

Efectos sobre el diseño de lenguajes. Las características de la entrada/salida interactiva son lo suficientemente diferentes de las operaciones ordinarias con archivos como para que casi todos los lenguajes proyectados para un entorno de procesamiento por lotes experimenten cierta dificultad para adaptarse a un entorno interactivo. Estas diferencias se analizan en la sección 4.3.12. El C, por ejemplo, incluye funciones para tener acceso a líneas de texto desde un archivo y otras funciones que alimentan directamente cada carácter conforme lo digita el usuario en una terminal. La introducción directa de texto desde una terminal en Pascal, sin embargo, suele ser muy engorrosa. Por esta razón, el C (y su derivado C++) se ha vuelto mucho más popular como lenguaje para escribir programas interactivos.

El manejo de errores en un entorno interactivo recibe un tratamiento diferente. Si se introducen mal los datos de entrada desde un teclado, el programa puede desplegar un mensaje de error y solicitar una corrección al usuario. Las características del lenguaje para manejar el error dentro del programa (por ejemplo, no tomándolo en cuenta e intentando continuar) tienen menos importancia. Sin embargo, la terminación del programa como respuesta a un error no es ordinariamente aceptable (a diferencia del procesamiento por lotes).

Los programas interactivos deben utilizar con frecuencia algún concepto de restricciones de tiempo. Por ejemplo, en un juego de video, la falta de respuesta a una escena desplegada dentro de un intervalo fijo de tiempo haría que el programa invocara cierta respuesta. Un programa interactivo que opera con tanta lentitud que no puede responder a un comando de entrada en un periodo razonable se suele considerar como inútil.

Por último, en un entorno interactivo el concepto de un programa principal suele estar ausente. En su lugar, el programa se compone de un conjunto de subprogramas y el usuario introduce el "programa principal" como una serie de comandos en la terminal. Así, la interacción con el usuario adopta la forma de una solicitud de un comando, seguida de la ejecución del mismo, seguida de la solicitud de otro comando y así sucesivamente. ML, LISP y Prolog exhiben este tipo de comportamiento.

1.4.3 Entornos de sistemas incrustados

Un sistema de computadora que se usa para controlar parte de un sistema más grande como una planta industrial, una aeronave, una máquina herramienta, un automóvil o incluso un tostador doméstico se conoce como un *sistema de computadora incrustado*. El sistema de computadora se ha vuelto parte integral del sistema más grande, y la falla del sistema de computadora significa también comúnmente la falla del sistema mayor. A diferencia de los dos entornos anteriores, el de procesamiento por lotes y el interactivo, donde la falla del programa suele ser simplemente un inconveniente y el programa se tiene que volver a ejecutar, con frecuencia la falla de una aplicación incrustada puede poner en peligro la vida, desde la avería de una computadora automotriz que hace que el auto se ahogue a altas velocidades en una carretera, a la falla de una computadora de a bordo que hace que un motor del avión se apague durante el despegue, a la descompostura de una computadora que hace que una planta nuclear se sobrecaliente, a la avería de una computadora de hospital que interrumpe la vigilancia de los pacientes, hasta la falla de su reloj digital que ocasiona que usted llegue tarde a una reunión.

La seguridad de funcionamiento y corrección son atributos primarios de los programas que se usan en estos dominios. De los lenguajes de la parte II, Ada, C y C++ se usan extensamente para satisfacer algunos de los requerimientos especiales de los entornos de sistema incrustado.

Efectos sobre el diseño de lenguajes. Los programas escritos para sistemas incrustados suelen operar sin un sistema operativo subyacente y sin los archivos de entorno y dispositivos de E/S usuales. En vez de ello, el programa debe interactuar directamente con dispositivos de E/S distintos de los normales a través de procedimientos especiales que toman en cuenta las peculiaridades de cada dispositivo. Por esta razón, los lenguajes para sistemas incrustados suelen hacer mucho menos énfasis en archivos y operaciones de entrada/salida orientadas a archivos. El acceso a dispositivos especiales se suele facilitar a través de características del lenguaje que proporcionan acceso a registros particulares del hardware, localidades de memoria, manejadores de interrupciones o subprogramas escritos en lenguaje ensamblador u otros lenguajes de bajo nivel.

El manejo de errores en sistemas incrustados tiene una importancia particular. Ordinariamente, cada programa debe estar preparado para manejar todos los errores en forma interna, adoptando acciones apropiadas para recuperarse y continuar. La interrupción, excepto en el caso de una avería catastrófica del sistema, no suele ser una alternativa aceptable, y por lo común no hay un usuario en el entorno que pueda proporcionar una corrección interactiva del error. El manejo de errores debe ser capaz de dar cuenta de las fallas de los componentes del sistema, además de los tipos comunes de errores causados por valores de datos erróneos. En el Ada, esta preocupación por el manejo interno de errores se manifiesta en las extensas capacidades para el manejo de excepciones.

Los sistemas incrustados deben operar casi siempre en *tiempo real*; es decir, la operación del sistema mayor dentro del cual está incrustada la computadora requiere que el sistema de computadora sea capaz de responder a entradas y producir salidas dentro de intervalos de tiempo estrechamente restringidos. Por ejemplo, una computadora que controle el vuelo de un avión debe responder con rapidez a los cambios de altitud o velocidad. La operación de estos programas en tiempo real requiere capacidades del lenguaje para monitorear intervalos de tiempo, responder a demoras de más de una cierta duración (que pueden indicar la avería de un componente del sistema), e iniciar y concluir acciones en tiempos determinados.

Por último, un sistema de computadora incrustado suele ser un *sistema distribuido*, compuesto de más de una computadora. El programa que se ejecuta en un sistema de esta clase se compone por lo general de un conjunto de tareas que operan simultáneamente, donde cada una controla o una parte del sistema. El programa principal, si lo hay, existe sólo para iniciar la ejecución de las tareas. Una vez iniciadas, estas tareas se ejecutan por lo común en forma simultánea e indefinida, puesto que sólo deben concluir cuando falla el sistema completo o se apaga por alguna razón.

1.4.4 Entornos de programación

Un *entorno de programación* es el que resulta familiar para casi todos los lectores de este libro. Es el entorno en el cual los programas se crean y se ponen a prueba, y tiende a tener menos influencia sobre el diseño de lenguajes que el entorno operativo en el cual se espera

que los programas se ejecuten. Sin embargo, se reconoce ampliamente que la producción de programas que operan de manera confiable y eficiente se simplifica mucho a través de un buen entorno de programación y de un lenguaje que permita el uso de buenas herramientas y prácticas de programación.

Un entorno de programación consiste primordialmente en un conjunto de herramientas de apoyo y un lenguaje de para invocarlas. Cada herramienta de apoyo es otro programa que el programador puede utilizar como ayuda durante una o más de las etapas de creación de un programa. Las herramientas típicas en un entorno de programación incluyen editores, depuradores, verificadores, generadores de datos de prueba e impresoras.

Efectos sobre el diseño de lenguajes. Los entornos de programación han afectado el diseño de los lenguajes principalmente en dos áreas importantes: las características que facilitan la compilación por separado y el ensamblado de un programa a partir de componentes, así como las características que ayudan a poner a prueba y depurar los programas.

Compilación por separado. Por lo común, en la construcción de cualquier programa grande es deseable que distintos programadores o grupos de programación proyecten, codifiquen y pongan a prueba partes del programa antes del ensamblado final de los componentes en un programa completo. Esto requiere que el lenguaje esté estructurado de manera que los subprogramas individuales u otras partes se puedan compilar y ejecutar por separado, sin las demás partes, y luego fusionarse sin cambio en el programa final.

La compilación por separado se dificulta por el hecho de que, al compilar un subprograma, el compilador puede necesitar información acerca de otros subprogramas u objetos de datos compartidos, como:

1. La *especificación* del número, orden y tipo de parámetros que espera cualquier subprograma llamado, permite al compilador comprobar si una invocación del subprograma externo es válida. También puede ser necesario conocer el lenguaje en el que el otro subprograma está codificado, para que el compilador pueda establecer la adecuada "secuencia de llamado" de instrucciones para transferir datos e información de control al subprograma externo durante la ejecución en la forma esperada por ese subprograma.
2. La *declaración de tipo de datos* para cualquier variable referida es necesaria para que el compilador pueda determinar la representación de almacenamiento de la variable externa, de modo que la referencia se pueda compilar usando la fórmula de acceso apropiada para la variable (por ejemplo, el desplazamiento correcto dentro del bloque de entorno común).
3. La *definición de un tipo de datos* que se define externamente pero se usa para declarar cualquier variable local dentro del subprograma se necesita para permitir al compilador asignar almacenamiento y computar fórmulas de acceso para datos locales.

Para suministrar esta información acerca de subprogramas compilados por separado, objetos de datos compartidos y definiciones de tipos, (1) el lenguaje puede requerir que la información se *redeclare* dentro del subprograma (en FORTRAN), (2) puede prescribir un *orden de compilación* particular para requerir que la compilación de cada subprograma vaya precedida de la compilación de la especificación de todos los subprogramas invocados y datos compartidos

(en Ada y hasta cierto punto en Pascal), o (3) puede requerir la presencia de una biblioteca que contenga las especificaciones pertinentes durante la compilación, de modo que el compilador pueda recuperarlas según lo necesite (en Ada y C++).

El término *compilación independiente* se usa por lo común para la opción (1). Cada subprograma se puede compilar de manera independiente sin información externa alguna; el subprograma es totalmente autosuficiente. La compilación independiente tiene la desventaja de que ordinariamente no hay manera de comprobar la congruencia de la información acerca de subprogramas y datos externos que se redeclaran en el subprograma. Si las declaraciones dentro del subprograma no concuerdan con la estructura real de los datos o subprograma externos, entonces aparece un error sutil en la etapa final de ensamblado que no habría sido detectado durante la puesta a prueba de las partes del programa compiladas en forma independiente.

Las opciones (2) y (3) requieren algún medio para dar o colocar en una biblioteca especificaciones de subprogramas, definiciones de tipos y entornos comunes antes de la compilación de un subprograma. Ordinariamente, es deseable permitir que se omita el *cuerpo* (variables y enunciados locales) de un subprograma y dar únicamente la *especificación*. El cuerpo se puede compilar por separado, más tarde. En Ada, por ejemplo, todos los subprogramas, tareas, o paquetes se dividen en dos partes, una especificación y un cuerpo, las cuales se pueden compilar por separado o colocarse en una biblioteca, según se requiera, para permitir la compilación de otros subprogramas. Una llamada de subprograma hecha a un subprograma que todavía no ha sido compilado se conoce como un *fragmento*. Un subprograma que contiene fragmentos se puede ejecutar y, cuando se llega a un fragmento, la llamada causa que se imprima un mensaje de diagnóstico de sistema (o que se adopte otra acción) en vez de una invocación real del subprograma. Así pues, un subprograma compilado por separado se puede ejecutar para fines de prueba, a pesar de que no está todavía disponible el código para algunas de las rutinas que él invoca.

Otro aspecto de la compilación por separado que afecta el diseño de lenguajes es en cuanto al uso de *nombres compartidos*. Si varios grupos están escribiendo partes de un programa grande, suele ser difícil asegurar que los nombres que usa cada grupo para subprogramas, entornos comunes y definiciones de tipos compartidas sean distintos. Un problema común es encontrar, durante el ensamblado del programa final completo, que varios subprogramas y otras unidades de programa tienen nombres iguales. Esto significa a menudo una revisión tediosa y lenta de código ya verificada. Los lenguajes emplean tres métodos para evitar este problema:

1. Todo nombre compartido, como en un enunciado *externo* en C, debe ser único, y es obligación del programador asegurar que así sea. Se deben adoptar *convenciones para asignación de nombres* desde un principio, de manera que cada grupo disponga de un conjunto definido de nombres que puede utilizar para subprogramas (a saber, "todos los nombres que use su grupo deben comenzar con QQ"). Por ejemplo, los nombres que se usan dentro de los archivos #include del C estándar llevan ordinariamente "_" como prefijo, de modo que los programadores deberán evitar nombres variables que comiencen con la línea de subrayado.

2. Los lenguajes suelen emplear reglas de *definición de* ámbito para ocultar nombres. Si un subprograma está contenido dentro de otro subprograma, los otros subprogramas

compilados por separado sólo conocen los nombres del subprograma más exterior. Los lenguajes como Pascal, C y Ada emplean este mecanismo. La estructura de bloques y el ámbito de los nombres se analizan en forma más completa en la sección 7.2.2.

3. Los nombres se pueden conocer agregando explícitamente sus definiciones desde una biblioteca externa. Éste es el mecanismo básico de *herencia* en lenguajes orientados a objetos. Al incluir en un subprograma una definición de clase externamente definida, se vuelven conocidos otros objetos que define esa clase, como en Ada y C++. En Ada, los nombres también pueden ser *homónimos*, de manera que varios objetos pueden tener el mismo nombre. En tanto el compilador pueda decidir a cuál objeto se hace efectivamente referencia, no es necesario hacer cambios en el programa de llamado. La homonimia se analiza más cabalmente en el capítulo 8.

Puesta a prueba y depuración. Casi todos los lenguajes contienen ciertas características que facilitan la puesta a prueba y la depuración de programas. Unos cuantos ejemplos representativos son:

1. *Características para rastreo de ejecución.* Prolog, LISP y muchos otros lenguajes interactivos suministran características que permiten marcar enunciados y variables particulares para "rastrearlos" durante su ejecución. Siempre que se ejecuta un enunciado marcado o se asigna un nuevo valor a una variable marcada, la ejecución del programa se interrumpe y se invoca un subprograma de rastreo designado (que típicamente imprime información de depuración apropiada).

2. *Puntos de interrupción.* En un entorno interactivo de programación, los lenguajes suelen suministrar una característica donde el programador puede especificar puntos del programa como *puntos de interrupción*. Cuando se alcanza un punto de interrupción durante la ejecución del programa, la misma se interrumpe y el control se traslada al programador en una terminal. El programador puede inspeccionar y modificar valores de variables y luego reiniciar el programa a partir del punto de interrupción.

3. *Asertos.* Un aserto es una expresión condicional que se inserta como un enunciado independiente en un programa, por ejemplo:

   aserto(X>0 **y** A=1) **o** (X=0 **y** A>B+10)

   El aserto expresa las relaciones que deben cumplirse entre los valores de las variables en ese punto del programa. Cuando el aserto se "habilita", el compilador inserta código en el programa compilado para poner a prueba las condiciones expresadas. Durante la ejecución, si las condiciones no se cumplen, entonces la misma se interrumpe y se invoca un manejador de excepciones para imprimir un mensaje o adoptar otra acción. Después de que el programa se depura, los asertos se pueden "inhabilitar" para que el compilador no genere código para su verificación. Entonces se convierten en comentarios útiles que ayudan en la documentación del programa. Se trata de un concepto sencillo que existe en varios lenguajes, entre ellos C++.

1.4.5 Marcos de ambiente

A lo largo de la última década, el concepto de ambientes de apoyo ha evolucionado. Un ambiente de apoyo consiste en servicios de infraestructura que se conocen como *marco de ambiente*. Este marco suministra servicios como un depósito de datos, interfaz gráfica de usuario, seguridad y servicios de comunicación. Los programas se escriben de modo que utilicen estos servicios. Los programadores tienen que ser capaces de usar fácilmente los servicios de infraestructura como parte de su diseño de programas. De acuerdo con esto, los lenguajes se proyectan a veces de modo que permitan un fácil acceso a estos servicios de infraestructura.

Por ejemplo, todos los programas escritos en los años sesenta incluían rutinas específicas de entrada/salida para manejar la comunicación con el usuario. Con el crecimiento de los sistemas interactivos, el concepto de ventanas desplegadas en una pantalla se ha convertido en el formato normal de salida. En la actualidad, un marco de entorno contendría un administrador de ventanas subyacente, como Motif, que utiliza el Sistema de Windows X, y el programa sólo necesitaría recurrir a funciones específicas de Motif para desplegar ventanas, menús y barras de recorrido, y para ejecutar casi todas las acciones comunes con ventanas. Las interfaces de Windows X son parte del marco de ambiente, el cual proporciona a todos los programas que lo usan un patrón común de comportamiento para el usuario en una terminal.

1.5 LECTURAS ADICIONALES SUGERIDAS

Son numerosos los textos que analizan lenguajes de programación particulares a un nivel introductorio. Los libros de [DERSHEM y JIPPING 1995], [LOUDEN 1993], [SEBESTA 1993] y [SETHI 1989] proporcionan puntos de vista alternativos de muchas de las cuestiones aquí tratadas. La historia inicial de los lenguajes de programación se trata en forma extensa en [SAMMET 1969, SAMMET 1972] y [ROSEN 1967]. Wexelblat [WEXELBLAT 1981] editó una colección de artículos de los diseñadores originales de muchos lenguajes importantes, los cuales también son de interés e incluyen ALGOL, COBOL, FORTRAN y PL/I. Una segunda conferencia sobre la "Historia de los lenguajes de programación" tuvo lugar en 1993 [ACM 1993]. Este volumen proporciona información adicional sobre la historia de Prolog, C, LISP, C++, Ada y Pascal, así como de varios otros lenguajes que no se estudian en este libro. El papel de la estandarización de lenguajes se analiza en [RADA y BERG 1995].

Todos los lenguajes que se analizan en la parte II muestran influencias de sus entornos de programación y operación. Ada, C, LISP, ML y Prolog son los que han sido más fuertemente influidos por estas consideraciones. Dos informes de NIST [BROWN *et al.* 1993, NIST 1993] describen modelos de servicios de infraestructura ambiental. [BROWN *et al.* 1992] y [PERRY y KAISER 1991] sintetizan muchos diseños modernos de ambiente. El papel de la corrección en el diseño de programas y las limitaciones de las técnicas de verificación se encuentran resumidas en [ABRAMS y ZELKOWITZ 1994]. *IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering* y *Software Practice and Experience* son tres revistas que suelen incluir artículos destacados en esta área.

1.6 PROBLEMAS

1. Para un lenguaje que se utilice intensamente en su sistema local de computadora, evalúe las razones de su éxito de acuerdo con la lista de criterios que se dan en la sección 1.3. ¿Se debería ampliar la lista?
2. Elija un lenguaje estandarizado para el cual usted tenga acceso al compilador. Escriba un programa no conformado que sin embargo se compile y ejecute. Enumere varias características no estándar que el compilador acepta.
3. Cuando se revisa la definición estándar de un lenguaje (por lo común cada 5 o 10 años), una influencia importante en la revisión es el requisito de hacer que los programas escritos en la versión anterior sean transportables a implementaciones de la versión revisada. Por consiguiente, durante la revisión existe presión para hacer que la nueva versión incluya características de la versión anterior. En cuanto a las diversas definiciones estándar de Ada, COBOL o FORTRAN, compare las características y enumere varias de ellas que, según su criterio, sean deficientes y cuya presencia en un estándar posterior se pueda atribuir principalmente al deseo de conservar la compatibilidad con el primero.
4. Considere el lenguaje simple siguiente. *a* y *b* representan nombres de variables enteras. Cada enunciado puede tener un rótulo como prefijo. Los enunciados en el lenguaje incluyen:

a = b	Asignar a a el valor de b
a = a+1	Sumar 1 a a
a = a–1	Restar 1 de a
si a=0 entonces ir a L	Si a=0 transferir el control al enunciado L
si a>0 entonces ir a L	Si a>0 transferir el control al enunciado L
ir a L	Transferir el control al enunciado L
alto	Detener la ejecución

Por ejemplo, el programa que computa $a = a + b$ puede estar dado por:

```
L:  a=a+1
    b=b–1
    si b>0 entonces ir a L
    alto
```

(a) Escriba los programas siguientes en este lenguaje:

(1) Dados a y b, computar $x = a + b$
(2) Dados a y b, computar $x = a \times b$
(3) Dados a, b, c y d, computar $x = a \times b$ y $y = c \times d$

(b) Exponga un conjunto mínimo de extensiones necesario para que este lenguaje sea fácil de usar. Tome en cuenta conceptos como subprogramas, enunciados nuevos, declaraciones, etcétera.

5. Las características del C permiten expresar el mismo significado de muchas maneras. ¿Cuántos enunciados diferentes puede usted escribir en C que sumen 1 a la variable X, es decir, que sean equivalentes a $X = X + 1$? Analice las ventajas y desventajas de este aspecto del diseño del C.

6. El enunciado **aserto** de la página 29 se puede implementar como una prueba de tiempo de ejecución que se aplica cada vez que se llega al enunciado, o puede ser una propiedad comprobada como verdadera en ese punto del programa.

   (a) Exponga cómo se puede implementar cada enfoque. ¿Cuál es la dificultad para implementar cada enfoque?

   (b) ¿Cuál es la "verdad" última de cada aserto para todas las ejecuciones de ese programa (es decir, cuándo será siempre verdadero ese aserto) para cada método de implementación?

7. Las primeras computadoras se proyectaron como calculadoras electrónicas para resolver problemas numéricos. Especule sobre cómo serían los lenguajes de programación si estas primeras máquinas se hubieran desarrollado para un propósito distinto (por ejemplo, procesamiento de texto, control de robots, juegos).

8. Suponga que un nuevo diseño de lenguaje provee tres tipos básicos de datos: *entero*, *real* y *de carácter*. También proporciona la capacidad para declarar *arreglos* y *registros* de datos. Los arreglos tienen elementos del mismo tipo y los registros los tienen de tipos mixtos. Use el concepto de *ortogonalidad* para criticar estas dos variantes del nuevo diseño:

   (a) Los elementos de arreglos y registros pueden ser de cualquiera de estos tipos básicos de datos y ellos mismos también pueden ser arreglos o registros (por ejemplo, un elemento de un registro puede ser un arreglo).

   (b) Los elementos de arreglos y registros pueden ser del tipo *entero* o *real*. Los arreglos de caracteres se llaman *cadenas* y reciben un tratamiento especial. Los elementos de registros pueden ser del tipo *de carácter*. Los registros pueden tener arreglos o elementos, pero los arreglos no pueden tener registros como elementos. Los arreglos no pueden tener arreglos como elementos, pero se suministran arreglos multidimensionales para obtener el mismo efecto.

Capítulo 3

Problemas de traducción de lenguajes

Durante los primeros días del diseño de lenguajes (es decir, durante la época formativa del desarrollo de FORTRAN, ALGOL, COBOL y LISP alrededor de 1960), se pensaba que una sintaxis formal era todo lo que se requería para especificar programas. El concepto de una gramática libre de contexto o gramática de forma de Backus-Naur (BNF, *Backus-Naur Form*) se desarrolló y se utilizó con éxito para especificar la sintaxis de un lenguaje (y se analiza más adelante en este capítulo). Aún constituye la técnica principal actual para describir los componentes de un programa.

3.1 SINTAXIS DE LENGUAJES DE PROGRAMACIÓN

La *sintaxis*, cuya definición sería "la disposición de palabras como elementos en una oración para mostrar su relación,"[1] describe la serie de símbolos que constituyen programas válidos. El enunciado en C: $X = Y + Z$ representa una serie válida de símbolos, en tanto que $XY + -$ no representa una secuencia válida de símbolos para un programa en C.

La sintaxis suministra información significativa que se necesita para entender un programa y proporciona información imprescindible para la traducción del programa fuente a un programa objeto. Por ejemplo, casi cualquier persona que lea este texto interpretará que la expresión $2 + 3 \times 4$ tiene el valor de 14 y no 20. Es decir, la expresión se interpreta como si estuviera escrita como $2 + (3 \times 4)$ y no como si fuera $(2 + 3) \times 4$. Se puede especificar una u otra interpretación, si se desea, por sintaxis y con ello guiar al traductor a la generación de las operaciones correctas para evaluar esta expresión.

Como en la expresión ambigua en español, "nada en el agua", el solo desarrollo de una sintaxis de lenguaje es insuficiente para especificar sin ambigüedad la estructura de un enunciado. En un enunciado como $X = 2.45 + 3.67$, la sintaxis no nos puede decir si se declaró la variable X, o si X se declaró como tipo real. Los resultados de $X = 5$, $X = 6$ y $X = 6.12$ son todos posibles si X y + denotan enteros, X denota un entero y + es adición de números reales, y X y + denotan valores reales, respectivamente. Se necesita algo más que sólo estructuras sintácticas para la plena descripción de un lenguaje de programación. Otros atributos, bajo el término general de *semántica*, como el uso de declaraciones, operaciones,

[1] *Webster's New World Dictionary*, Fawcett, 1979.

control de secuencia y entornos de referimiento, afectan a una variable y no siempre están determinados por reglas de sintaxis.

Aunque las descripciones sintácticas de un lenguaje de programación son insuficientes para la comprensión completa del lenguaje, son atributos importantes en esta descripción. En su mayor parte, la descripción de la sintaxis es un "problema resuelto". Cuando se analice el diseño de traductores más adelante en este capítulo, veremos que la primera fase de la comprensión sintáctica del programa fuente es bastante mecánica. Las herramientas como YACC (*Yet Another Compiler Compiler*; todavía otro compilador de compiladores) producen automáticamente esta descripción sintáctica de un programa dado. La aplicación de la comprensión semántica para generar un programa objeto eficiente a partir de un programa fuente dado requiere todavía mucha habilidad, así como la aplicación de ciertos métodos formales al proceso.

3.1.1 Criterios generales de sintaxis

El propósito primordial de la sintaxis es proveer una notación para la comunicación entre el programador y el procesador de lenguajes de programación. Sin embargo, la elección de estructuras sintácticas particulares está restringida sólo ligeramente por la necesidad de comunicar elementos particulares de información. Por ejemplo, el hecho de que una variable particular tenga un valor de tipo *número real* se puede representar en cualquiera de una docena de formas diferentes en un programa, a través de una declaración explícita como en Pascal a través de una *convención de nomenclatura* implícita, como en FORTRAN, y así sucesivamente. Los detalles de sintaxis se eligen en gran medida con base en criterios secundarios, como la legibilidad, los cuales no tienen relación con el objetivo primario de comunicar información al procesador de lenguajes.

Existen muchos criterios secundarios, pero se pueden clasificar en forma aproximada bajo los objetivos generales de hacer que los programas sean fáciles de leer, fáciles de escribir, fáciles de traducir y no ambiguos. Consideraremos a continuación algunas de las formas de designar estructuras sintácticas de lenguajes para satisfacer estos objetivos a menudo en conflicto.

Legibilidad. Un programa es legible si la estructura subyacente del algoritmo y los datos que el programa representa quedan de manifiesto al inspeccionar el texto del programa. Se suele decir que un programa legible es *autodocumentable*, es entendible sin documentación independiente alguna (aunque este objetivo se alcanza pocas veces en la práctica). La legibilidad se mejora a través de características de lenguaje tales como formatos naturales de enunciado, enunciados estructurados, uso abundante de palabras clave y palabras pregonadas, recursos para comentarios incrustados, identificadores de longitud sin restricción, símbolos nemotécnicos de operadores, formatos de campo libre y declaraciones completas de datos. Desde luego, la legibilidad no se puede garantizar mediante el diseño de un lenguaje, porque incluso el mejor diseño se puede entrampar por una programación deficiente. Por otra parte, el diseño sintáctico puede obligar incluso al programador mejor intencionado a escribir programas ilegibles (como suele ocurrir en el APL). El diseño del COBOL hace un fuerte énfasis en la legibilidad, con frecuencia a expensas de la facilidad de escritura y de traducción.

La legibilidad mejora a través de una sintaxis de programa en la cual las diferencias sintácticas reflejan diferencias sintácticas subyacentes, de modo que las construcciones de programa que hacen cosas similares se ven parecidas, y las construcciones de programa que hacen cosas radicalmente distintas se ven diferentes. En general, cuanto mayor es la variedad de construcciones sintácticas empleadas, resulta más fácil hacer que la estructura del programa refleje estructuras semánticas subyacentes distintas.

Los lenguajes que suministran sólo unas pocas construcciones sintácticas conducen en general a programas menos legibles. En APL o SNOBOL4, por ejemplo, sólo se provee un formato de enunciado. Las diferencias entre un enunciado de asignación, una llamada de subprograma, un enunciado **goto** simple, una devolución de subprograma, una bifurcación condicional de múltiples vías, y otras estructuras comunes de programa se reflejan sintácticamente sólo por diferencias en un solo o unos pocos símbolos de operador dentro de una expresión compleja. Suele ser necesario un análisis minucioso de un programa para determinar incluso su estructura de control a *grosso modo*. Más aún, un simple error de sintaxis, como un solo carácter incorrecto en un enunciado, puede alterar en forma radical el significado del enunciado sin hacerlo sintácticamente incorrecto. En LISP, los errores de concordancia de paréntesis causan problemas similares; una de las extensiones de Scheme a LISP sirve para evitar este problema.

Facilidad de escritura. Las características sintácticas que hace que un programa sea fácil de escribir suelen hallarse en conflicto con las características que facilitan su lectura. El atributo de fácil escritura mejora a través del uso de estructuras sintácticas concisas y regulares, en tanto que para la legibilidad son de ayuda diversas construcciones más elocuentes. C tiene por desgracia este atributo de proveer recursos para programas muy concisos de difícil lectura, aunque cuenta con un complemento detallado de características útiles.

Las convenciones sintácticas implícitas que permiten dejar sin especificar declaraciones y operaciones hacen a los programas más cortos y fáciles de escribir pero dificultan su lectura. Otras características favorecen ambos objetivos; por ejemplo, el uso de enunciados estructurados, formatos simples de enunciados naturales, símbolos nemotécnicos de operaciones e identificadores no restringidos facilita por lo común la escritura de programas al permitir que la estructura natural de los algoritmos y datos del problema se represente directamente en el programa.

Una sintaxis es *redundante* si comunica el mismo elemento de información en más de una forma. Cierta redundancia es útil en la sintaxis de lenguajes de programación porque facilita la lectura del programa y también permite revisar en busca de errores durante la traducción. La desventaja es que la redundancia hace a los programas más elocuentes y por tanto dificulta su escritura. Casi todas las reglas por omisión para el significado de construcciones de lenguaje buscan reducir la redundancia eliminando la expresión explícita de significados que se pueden inferir del contexto. Por ejemplo, en vez de requerir la declaración explícita del tipo de cada variable simple, FORTRAN suministra una convención de nomenclatura que declara implícitamente que las variables cuyos nombres comienzan con una de las letras I-N son de tipo entero y todas las demás variables son de tipo real. La mera incidencia de un nuevo nombre de variable en un programa basta para "declararla". Desafortunadamente, la conveniencia adicional queda neutralizada por un efecto negativo más serio: el compilador de FORTRAN no puede detectar un nombre de variable mal escrito. Si el programa utiliza un *ÍNDICE* va-

riable al que en algún punto se hace referencia como *ÍNDCE*, el compilador supone que *ÍNDCE* es una nueva variable entera y se introduce un error sutil en el programa. Si se requiere una declaración explícita para cada variable, como en Pascal, entonces el compilador caracteriza el nombre de variable mal escrito como un error. A causa de este efecto de enmascarar errores en los programas, los programas que carecen de toda redundancia suelen ser difíciles de usar.

Facilidad de verificación. Tiene relación con la legibilidad y facilidad de escritura el concepto de corrección del programa o *verificación del programa.* Luego de muchos años de experiencia, ahora comprendemos que, si bien entender cada enunciado de lenguaje de programación es relativamente fácil, el proceso global de crear programas correctos es en extremo difícil. Por consiguiente, se necesitan técnicas que permitan probar que el programa es matemáticamente correcto. Esto se analiza más a fondo en el capítulo 9.

Facilidad de traducción. Un tercer objetivo en conflicto es el de hacer que los programas sean fáciles de traducir a una forma ejecutable. Legibilidad y facilidad de escritura son criterios dirigidos a las necesidades del programador humano. La facilidad de traducción se relaciona con las necesidades del traductor que procesa el programa escrito. La clave para una traducción fácil es la regularidad de la estructura. La sintaxis del LISP proporciona un ejemplo de una estructura de programa que no es particularmente legible ni especialmente fácil de escribir pero que resulta en extremo fácil de traducir. La estructura sintáctica completa de cualquier programa en LISP se puede describir en unas cuantas reglas sencillas a causa de la regularidad de la sintaxis. La traducción de los programas se dificulta conforme aumenta el número de construcciones sintácticas especiales. Por ejemplo, la traducción de COBOL se hace muy difícil a causa del gran número de formas de enunciados y declaraciones que se permiten, no obstante que la semántica del lenguaje no es complicada de manera expresa.

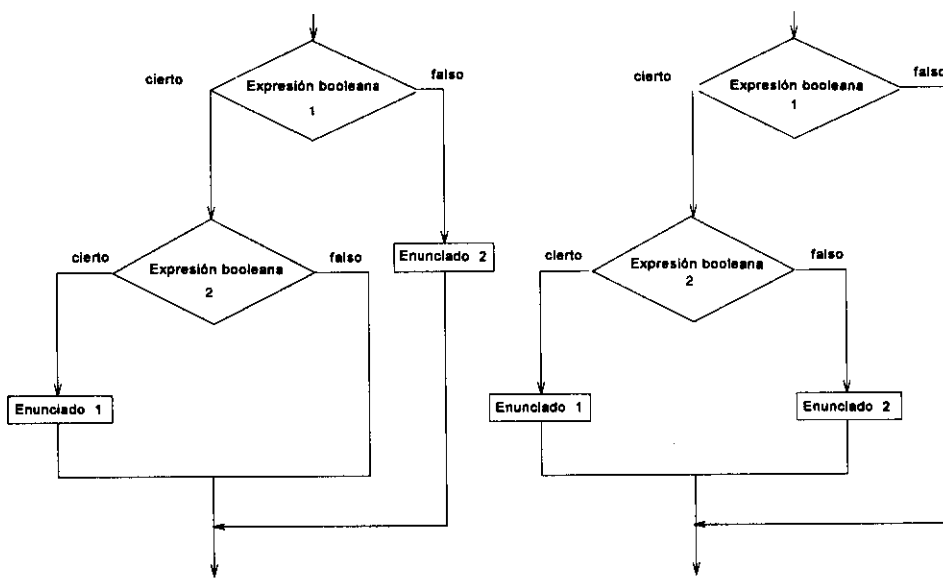


Figura 3.1. Dos interpretaciones de un enunciado condicional.

Carencia de ambigüedad. La ambigüedad es un problema medular en todo diseño de lenguaje. Una definición de lenguaje proporciona idealmente un significado único para cada construcción sintáctica que el programador puede escribir. Una construcción ambigua permite dos o más interpretaciones distintas. El problema de ambigüedad surge por lo común no en la estructura de elementos individuales de programa, sino en la interacción entre diferentes estructuras. Por ejemplo, tanto Pascal como ALGOL permiten dos formas distintas de enunciado condicional:

1. **if** *expresión booleana* **then** *enunciado*$_1$ **else** *enunciado*$_2$
2. **if** *expresión booleana* **then** *enunciado*$_1$

La interpretación que se debe dar a cada forma de enunciado está claramente definida. Sin embargo, cuando las dos formas se combinan al permitir que el *enunciado*$_1$ sea otro enunciado condicional, entonces se forma la estructura que se conoce como *else ambiguo*:

if *expresión booleana*$_2$ **then if** *expresión booleana*$_2$ **then** *enunciado*$_1$ **else** *enunciado*$_2$

Esta forma de enunciado es ambigua porque no queda claro cuál de las dos secuencias de ejecución de la figura 3.1 es la deseada. La sintaxis de FORTRAN ofrece otro ejemplo. Una referencia a A(I,J) podría ser ya sea una referencia a un elemento del arreglo bidimensional A o una llamada del subprograma de funciones A porque la sintaxis de FORTRAN para llamadas de función y referencias a arreglos es la misma. Surgen ambigüedades similares en casi todos los lenguajes de programación.

De hecho, las ambigüedades en FORTRAN y ALGOL antes mencionadas han sido resueltas en ambos lenguajes. En el enunciado condicional en ALGOL, la ambigüedad se ha resuelto cambiando la sintaxis del lenguaje para introducir un par delimitador **begin... end** requerido en torno al enunciado condicional incrustado. Así, la combinación natural pero ambigua de dos enunciados condicionales ha sido reemplazada por las dos construcciones menos naturales pero no ambiguas, de acuerdo con la interpretación deseada.

1. **if** *expresión booleana*$_2$ **then begin if** *expresión booleana*$_2$ **then** *enunciado*$_1$ **end else** *enunciado*$_2$
2. **if** *expresión booleana*$_2$ **then begin if** *expresión booleana*$_2$ **then** *enunciado*$_1$ **else** *enunciado*$_2$ **end**

Se emplea una solución más sencilla en Ada: cada enunciado **if** debe terminar con el delimitador **end if**. En C y Pascal se usa otra técnica para resolver la ambigüedad: se elige una interpretación arbitraria para la construcción ambigua, en este caso el **else** final se aparea con el **then** más próximo para que el enunciado combinado tenga el mismo significado que la segunda de las construcciones precedentes en ALGOL. La ambigüedad de las referencias a funciones y arreglos en FORTRAN se resuelve mediante la regla siguiente: se supone que la construcción $A(I,J)$ es una llamada de función si no se da una declaración para un arreglo A. Puesto que cada arreglo se debe declarar antes de su uso en un programa, el traductor puede verificar fácilmente si existe en efecto un arreglo A al cual se aplica la referencia. Si no lo encuentra, entonces el traductor supone que la construcción es una llamada a una función

externa *A*. Esta suposición no se puede comprobar sino hasta el tiempo de carga, cuando todas las funciones externas (incluidas las funciones de biblioteca) se vinculan en el programa ejecutable final. Si el cargador no encuentra una función *A*, entonces se produce un mensaje de error del cargador. En Pascal se utiliza una técnica distinta para distinguir llamadas de función de referencias a arreglos: se hace una distinción sintáctica. Se emplean paréntesis rectangulares para encerrar listas de subíndices en referencias a arreglos (por ejemplo, *A*[*I*,*J*]) y se usan paréntesis normales para encerrar listas de parámetros de llamadas de función (por ejemplo, *A*(*I*,*J*)).

3.1.2 Elementos sintácticos de un lenguaje

El estilo sintáctico general de un lenguaje está dado por la selección de diversos elementos sintácticos básicos. Consideraremos en forma breve los más destacados de entre ellos.

Conjunto de caracteres. La elección del conjunto de caracteres es la primera que se hace al proyectar una sintaxis de lenguaje. Existen varios conjuntos de caracteres de uso amplio, como el conjunto ASCII, cada uno con un conjunto diferente de caracteres especiales además de las letras y dígitos básicos. Por lo común se elige uno de estos conjuntos estándar, aunque en ocasiones se puede usar un conjunto de caracteres especial, no estándar, como, por ejemplo, en el APL. La elección del conjunto de caracteres es importante en la determinación del tipo de equipo de entrada y salida que se puede usar al implementar el lenguaje. Por ejemplo, el conjunto básico de caracteres de C está disponible en casi todos los equipos de entrada y salida. El conjunto de caracteres de APL, por otra parte, no se puede usar directamente en la mayoría de los dispositivos de entrada/salida.

El uso general de bytes de 8 bits para representar caracteres parecía una opción razonable cuando la industria de computadoras pasó de caracteres de 6 bits a los de 8 bits a principios de la década de 1960. Doscientos cincuenta y seis caracteres parecían más que suficientes para representar las 52 letras mayúsculas y minúsculas, 10 dígitos y unos cuantos símbolos de puntuación. Sin embargo, en la actualidad la industria de computadoras es mucho más internacional. No muchos (de hecho, pocos) países usan las mismas 26 letras del idioma inglés. El español agrega la tilde (~), el francés usa acentos (´), y otros caracteres están presentes en ciertos idiomas (por ejemplo, å, ß, ö). Además, existen idiomas como el griego, hebreo y árabe con conjuntos de caracteres totalmente distintos. La representación de idiomas como el chino o el japonés, donde cada ideograma representa una palabra o frase, requiere un conjunto de caracteres del orden de 10 000 símbolos. Es evidente que los caracteres de un solo byte no funcionan en todos estos casos, así que los implementadores de lenguajes tienen que considerar cada vez más las representaciones de 16 bits (que son 65,536) para el conjunto de caracteres.

Identificadores. La sintaxis básica para identificadores, una cadena de letras y dígitos que comienza con una letra, tiene amplia aceptación. Las variaciones entre lenguajes se dan principalmente en la inclusión opcional de caracteres especiales como . o - para mejorar la legibilidad y en restricciones de longitud. Las restricciones de longitud, como la antigua restricción de FORTRAN a seis caracteres, obligan al uso de identificadores con poco valor nemotécnico en muchos casos y en esta forma restringen la legibilidad del programa en forma significativa.

Símbolos de operadores. Casi todos los lenguajes emplean los caracteres especiales + y – para representar las dos operaciones aritméticas básicas, pero más allá de esto casi no existe uniformidad. Las operaciones primitivas se pueden representar cabalmente por caracteres especiales, como se hace en APL. Alternativamente, se pueden usar identificadores para todas las primitivas, como en LISP, PLUS, TIMES, etcétera. Casi todos los lenguajes adoptan alguna combinación y utilizan caracteres especiales para ciertos operadores, identificadores para otros, y con frecuencia también algunas cadenas de caracteres que no encajan en estas dos categorías (por ejemplo, .EQ. y ** de FORTRAN, para igualdad y exponenciación, respectivamente).

Palabras clave y palabras reservadas. Una *palabra clave* es un identificador que se usa como una parte fija de la sintaxis de un enunciado, por ejemplo, "IF" al principio de un enunciado condicional en FORTRAN o "DO" al comenzar un enunciado de iteración en FORTRAN. Una palabra clave es una *palabra reservada* si no se puede usar también como un identificador elegido por el programador. Casi todos los lenguajes emplean actualmente palabras reservadas, con lo cual se mejora la capacidad de detección de errores de los traductores. La mayoría de los enunciados comienzan con una palabra clave que designa el tipo de enunciado: READ, IF, WHILE, etcétera.

El análisis sintáctico durante la traducción se facilita usando palabras reservadas. El análisis sintáctico en FORTRAN, por ejemplo, se dificulta por el hecho de que un enunciado que comienza con DO o IF puede no ser en realidad un enunciado de iteración o condicional. Puesto que DO e IF no son palabras reservadas, el programador las puede elegir legítimamente como nombres de variables. COBOL emplea palabras reservadas en abundancia, pero existen tantos identificadores reservados que es difícil recordarlos todos y, como consecuencia, uno suele elegir involuntariamente un identificador reservado como nombre de variable. Sin embargo, la dificultad primordial con las palabras reservadas se presenta cuando se necesita ampliar el lenguaje para incluir enunciados nuevos empleando palabras reservadas, por ejemplo, como cuando el COBOL se revisa periódicamente para elaborar un estándar actualizado. La adición de una nueva palabra reservada al lenguaje significa que todo programa antiguo que utilice ese identificador como nombre de variable (u otro nombre) ya no es sintácticamente correcto, no obstante que no ha sido modificado.

Palabras pregonadas. Las *palabras pregonadas* son palabras opcionales que se insertan en los enunciados para mejorar la legibilidad. COBOL ofrece muchas opciones de este tipo. Por ejemplo, en el enunciado **goto**, que se escribe GO TO *rótulo*, se requiere la palabra clave GO, pero TO es opcional; no transmite información y sólo se usa para mejorar la legibilidad.

Comentarios. La inclusión de comentarios en un programa es una parte importante de su documentación. Un lenguaje puede permitir comentarios de varias maneras: (1) renglones de comentarios por separado en el programa, como en FORTRAN; (2) delimitados por marcadores especiales, como los /* y */ de C sin que importen los límites de renglón; o (3) comenzando en cualquier punto de un renglón pero ya concluidos al final del mismo, como el -- en Ada, / / en C++ o ! en FORTRAN 90. La tercera alternativa incluye a la primera y también permite introducir un comentario breve después del enunciado o declaración en un renglón. La segunda alternativa tiene la desventaja de que la falta de un delimitador de conclusión en un comentario transforma los enunciados siguientes (hasta el final del siguiente comentario)

en "comentarios", de modo que, aunque parezcan correctos al leer el programa, de hecho no se traducen ni se ejecutan.

Espacios en blanco. Las reglas sobre el uso de espacios en blanco varían ampliamente entre los lenguajes. En FORTRAN, por ejemplo, los espacios en blanco no son significativos en cualquier parte excepto en datos literales de cadena de caracteres. Otros lenguajes usan espacios en blanco como separadores, de modo que desempeñan un papel sintáctico importante. En SNOBOL4 la operación primitiva de concatenación se representa con un espacio en blanco, el cual también se usa como separador entre elementos de un enunciado (lo que ocasiona mucha confusión). En C, los espacios en blanco se pasan generalmente por alto, pero no siempre. En las primeras versiones de C, el símbolo = + era un solo operador, en tanto que = . + representaba dos operaciones. Para impedir este tipo de errores, la definición actual de C emplea el símbolo + = como un operador combinado, puesto que + = sería un error de sintaxis.

Delimitadores y corchetes. Un *delimitador* es un elemento sintáctico que se usa simplemente para señalar el principio o el final de alguna unidad sintáctica, como un enunciado o expresión. Los *corchetes* son delimitadores apareados, por ejemplo, paréntesis o parejas de **begin ... end**. Los delimitadores se pueden usar simplemente para mejorar la legibilidad o simplificar el análisis sintáctico, pero con más frecuencia tienen el importante propósito de eliminar ambigüedades definiendo explícitamente los límites de una construcción sintáctica particular.

Formatos de campos libres y fijos. Una sintaxis es de *campo libre* si los enunciados de programa se pueden escribir en cualquier parte de un renglón de entrada sin que importe la posición sobre el renglón o las interrupciones entre renglones. Una sintaxis de *campo fijo* utiliza la posición sobre un renglón de entrada para transmitir información. La sintaxis de campo fijo estricta, donde cada elemento de un enunciado debe aparecer dentro de una parte dada de un renglón de entrada, se observa más a menudo en lenguajes ensambladores. Es más común el uso de un formato de campo fijo parcial; por ejemplo, en FORTRAN los cinco primeros caracteres de cada renglón están reservados para un rótulo de enunciado. A veces se da un significado especial al primer carácter de un renglón de entrada; por ejemplo, en el SNOBOL4 los rótulos de enunciado, comentarios y renglones de continuación se distinguen por un carácter en la primera posición de un renglón. La sintaxis de campo fijo es cada vez más rara en la actualidad y la norma es el campo libre.

Expresiones. Las expresiones son funciones que acceden a objetos de datos en un programa y devuelven algún valor. Las expresiones son los bloques sintácticos básicos de construcción a partir de los cuales se construyen enunciados (y a veces programas). En lenguajes imperativos como C, las expresiones constituyen las operaciones básicas que permiten que cada enunciado modifique el estado de máquina. En lenguajes aplicativos como ML o LISP, las expresiones forman el control básico de secuencia que dirige la ejecución de programas. Se estudiará más a fondo la construcción de expresiones en el capítulo 6.

Enunciados. Los enunciados constituyen el componente sintáctico más destacado en los lenguajes imperativos, la clase dominante de lenguajes que se usan en la actualidad. Su sintaxis tiene un efecto decisivo sobre la regularidad, legibilidad y facilidad de escritura generales del lenguaje. Ciertos lenguajes adoptan un formato único de enunciado básico, mientras que

otros emplean diferente sintaxis para cada tipo distinto de enunciado. El primer enfoque hace énfasis en la regularidad, en tanto que el segundo resalta la legibilidad. El SNOBOL4 tiene sólo una sintaxis básica de enunciados, el enunciado de sustitución por concordancia de patrones, a partir del cual se pueden derivar otros tipos de enunciados omitiendo elementos del enunciado básico. Casi todos los lenguajes se inclinan hacia el otro extremo de suministrar diferentes estructuras sintácticas para cada tipo de enunciado. COBOL es el más notable en este sentido. Cada enunciado en COBOL tiene una estructura peculiar en la que participan palabras clave especiales, palabras pregonadas, construcciones alternativas, elementos optativos, etc. La ventaja de usar diversas estructuras sintácticas, por supuesto, es que se puede hacer que cada una exprese de manera natural las operaciones que intervienen.

Una diferencia más importante en las estructuras de enunciado es la que existe entre los enunciados *estructurados* o *anidados* y los enunciados *simples*. Un enunciado simple no contiene otros enunciados incrustados. APL y SNOBOL4 sólo permiten enunciados simples. Un enunciado estructurado puede contener enunciados incrustados. Las ventajas de los enunciados estructurados se analizan extensamente en el capítulo 6.

3.1.3 Estructura de conjunto de programas y subprogramas

La organización sintáctica global de las definiciones de programa principal y subprogramas es tan variada como los otros aspectos de la sintaxis de los lenguajes. Los lenguajes de la parte II ilustran varias de las estructuras más comunes.

Definiciones individuales de subprogramas. FORTRAN ilustra una organización de conjunto en la cual cada definición de subprograma se trata como una unidad sintáctica individual. Cada subprograma se compila por separado y los programas compilados se vinculan durante la carga. El efecto de esta organización se hace particularmente manifiesto cuando se requiere que cada subprograma contenga declaraciones completas de todos los elementos de datos, incluso para aquellos que están en bloques COMUNES (COMMON) y se comparten con otros subprogramas. Estas declaraciones son necesarias porque se supone una compilación por separado. La unidad básica de subprograma representa una estructura que por lo general proporciona funcionalidad afín. Por ejemplo, los subprogramas podrían representar operaciones de arreglos, operaciones de interfaz de usuario o alguna función interna de procesamiento de datos.

Definiciones individuales de datos. Un modelo alternativo consiste en agrupar todas las operaciones que manipulan un objeto de datos determinado. Por ejemplo, un subprograma puede consistir en todas las operaciones que se ocupan de un formato específico de datos dentro del programa, operaciones para crear el registro de datos, operaciones para imprimir el registro de datos y operaciones para cómputo con el registro de datos. Este es el enfoque general del mecanismo de **class** (clase) en lenguajes como C++ y Smalltalk.

Definiciones de subprograma anidadas. El Pascal ilustra una estructura de programa anidada donde las definiciones de subprograma aparecen como declaraciones dentro del programa principal y ellas mismas pueden contener otras definiciones de subprograma anidadas dentro de sus definiciones a cualquier profundidad. Esta organización de conjunto del programa se

relaciona con el énfasis que se hace en Pascal en los enunciados estructurados, pero, de hecho, satisface un propósito distinto. Los enunciados estructurados se introducen en primer término para suministrar una notación natural para las divisiones jerárquicas comunes en la estructura de los algoritmos, pero las definiciones de subprograma anidadas sirven en vez de ello para proveer un entorno no local de referimiento para subprogramas que se define durante la compilación y que, en esta forma, permite la verificación estática de datos y la compilación de código ejecutable eficiente para subprogramas que contienen referencias no locales. Sin el anidamiento de definiciones de subprograma, es necesario ya sea proporcionar declaraciones para variables no locales dentro de cada definición de subprograma (como se hace en FORTRAN) o diferir toda la verificación de tipos para referencias no locales hasta el tiempo de ejecución. El anidamiento también cumple la menos importante función de permitir que los nombres de subprograma tengan un alcance menor que el global.

Definiciones individuales de interfaz. La estructura de FORTRAN permite la fácil compilación de subprogramas individuales, pero tiene la desventaja de que los datos que se usan a través de diferentes subprogramas pueden tener definiciones distintas que el compilador no podrá detectar durante la compilación. Por otra parte, Pascal permite al compilador tener acceso a todas estas definiciones para ayudar a encontrar errores, pero tiene la desventaja de que el programa completo, incluso si tiene una longitud de muchos miles de enunciados, se debe volver a compilar cada vez que se necesita cambiar un solo enunciado. C, ML y Ada utilizan aspectos de las dos técnicas anteriores para mejorar el comportamiento de compilación.

En estos lenguajes, una *implementación* de lenguaje consiste en varios subprogramas que deben interactuar juntos. Todos estos componentes, llamados *módulos*, se enlazan entre sí, como en FORTRAN, para crear un programa ejecutable, pero sólo es necesario volver a compilar cada vez los componentes que han cambiado. Por otra parte, los datos que se pasaron entre los procedimientos de un componente deben tener declaraciones comunes como en Pascal, lo que permite su verificación eficiente por parte del compilador. Sin embargo, para pasar información entre dos componentes compilados por separado, se requieren datos adicionales. Esto es manejado por un componente de *especificación* del programa. En C, el enfoque consiste en incluir ciertas operaciones de archivo de sistema operativo en el lenguaje permitiendo que el programa incluya archivos que contengan estas definiciones de interfaz. Los archivos ".h" de C constituyen los componentes de especificación y los archivos ".c" del programa fuente son los componentes de implementación.[2] En Ada, el planteamiento consistió en integrar estas características directamente en el lenguaje. Los programas se definen en componentes llamados *paquetes*, los cuales contienen ya sea la especificación de las definiciones de interfaz o la implementación del programa fuente (en el *cuerpo* del paquete) que va a usar estas definiciones.

Descripciones de datos separadas de enunciados ejecutables. COBOL contiene una forma anticipada de estructura de componentes. En un programa en COBOL, las declaraciones de datos y los enunciados ejecutables para todos los subprogramas se dividen en *divisiones de datos* y *divisiones de procedimientos* de programa individuales. Una tercera *división de entorno*

[2] El popular programa *make* de UNIX se usa para determinar cuáles archivos ".c" y ".h" han sido alterados para recompilar sólo los componentes del sistema que han cambiado.

Sinopsis de lenguaje 3.1: BASIC

Características: Un lenguaje con sintaxis y semántica extremadamente simples; enunciados numerados, nombres de variable son letra sola más dígito, vinculación simple de IF, iteración FOR y GOSUB (subrutina).

Historia: El BASIC (Beginner's All-purpose Symbolic Instruction Code) fue desarrollado en Dartmouth College por Thomas Kurtz y John Kemeny a principios de los años sesenta. Los objetivos eran suministrar un entorno sencillo de computación, en especial para estudiantes no científicos. Además, para aumentar la eficacia de la computación, el BASIC se proyectó como un lenguaje interactivo años antes de que el tiempo compartido se volviera la norma en la arquitectura de sistemas.
El BASIC también es un buen ejemplo de la autocontradicción en programación. Los críticos dicen que "BASIC" es la respuesta a la pregunta: ¿Qué es un oxímoron (frase contradictoria) de una palabra? Mientras que la sintaxis es extremadamente fácil de aprender, la complejidad de armar un programa hace al código de BASIC ilegible si se usa más de alrededor de una página de enunciados fuente. Por esta razón, las versiones más recientes de BASIC han incluido nombres de variables más largos, nombres de subrutinas y mayor flexibilidad en estructuras de control, lo que hace que BASIC se parezca más a lenguajes como Pascal o FORTRAN que al sencillo lenguaje desarrollado en los años sesenta.

Ejemplo: Calcule la suma de $1^2 + 2^2 + \ldots + 10^2$

```
100 REMARK S IS SUM; I IS INDEX
200 LET S=0
300 FOR I=1 TO 10
400 LET S = S + I * I
500 NEXT I
600 REMARK NEXT IS END OF LOOP
700 PRINT "SUM IS", S
800 STOP
900 REMARK OTHER STATEMENTS: IF S>400 THEN 200 - (bifurcar a 200)
1000 REMARK OTHER STATEMENTS: DIM A(20) - Un arreglo de 20
1100 REMARK OTHER STATEMENTS: GOSUB 100; RETURN - Subroutines
1200 REMARK OTHER STATEMENTS: READ A - Entrada
```

Referencia: T. E. Kurtz, "BASIC," *ACM History of Programming Languages Conference*, Los Ángeles (Junio de 1978) (*SIGPLAN Notices (13)8 [Agosto 1978]*), 103-118.

consiste en declaraciones que conciernen al entorno externo de operación. La división de procedimientos de un programa se organiza en subunidades que corresponden a cuerpos de subprograma, pero todos los datos son globales para todos los subprogramas, y nada hay que corresponda a los datos locales comunes de un subprograma. La ventaja de la división centralizada de datos que contiene todas las declaraciones de datos, es que hace valer la independencia lógica de los formatos de datos y los algoritmos de la división de procedimientos. También es conveniente tener las descripciones de datos reunidas en un lugar en vez de dispersas por todos los subprogramas.

Definiciones de subprograma no separadas. SNOBOL4 representa el extremo en cuanto a organización (o falta de organización) de conjunto de programas. No se hace distinción sintáctica alguna en SNOBOL4 entre los enunciados del programa principal y los enunciados de subprograma. Un programa, sin considerar el número de subprogramas que contiene, es sintácticamente sólo una lista de enunciados. Los puntos donde los subprogramas comienzan y terminan no se distinguen sintácticamente. Los programas simplemente se ejecutan y la ejecución de una llamada de función inicia un nuevo subprograma; la ejecución de una función RETURN (devolver) termina un subprograma. El comportamiento del programa es totalmente dinámico. De hecho, cualquier enunciado puede ser parte del programa principal y también parte de cualquier número de subprogramas al mismo tiempo, en el sentido de que se puede ejecutar en un punto durante la ejecución del programa principal y más tarde volverse a ejecutar como parte de la ejecución de un subprograma. Esta organización más bien caótica del programa es valiosa sólo en cuanto a que permite traducción en tiempo de ejecución y ejecución de nuevos enunciados y subprogramas con mecanismos relativamente sencillos.

El BASIC [KURTZ 1978] es otro ejemplo que va en contra de algunos de estos objetivos del diseño de lenguajes de programación. Desarrollado en Dartmouth College en los años sesenta, su meta principal era llevar la computación en una forma fácil al no científico. Como tal, tenía una sintaxis extremadamente sencilla. Si bien los programas largos en BASIC son en extremo difíciles de entender, es un lenguaje muy eficaz para construir pequeños programas "desechables" que es necesario ejecutar unas cuantas veces y luego se descartan. La sinopsis de Lenguaje 3.1 sintetiza el lenguaje BASIC. De los años sesenta a la fecha, sin embargo, las versiones sucesivas de BASIC se han vuelto más complejas y con frecuencia alcanzan la estructura de lenguajes como Pascal en cuanto a poder, sintaxis y complejidad.

3.2 ETAPAS DE TRADUCCIÓN

El proceso de traducción de un programa, de su sintaxis original a una forma ejecutable, es medular en toda implementación de lenguajes de programación. La traducción puede ser bastante sencilla, como en el caso de los programas en Prolog o LISP, pero, con frecuencia, el proceso puede ser bastante complejo. Casi todos los lenguajes se podrían implementar con sólo una traducción trivial si uno estuviera dispuesto a escribir un intérprete de *software* y a aceptar velocidades lentas de ejecución. En la mayoría de los casos, sin embargo, la ejecución eficiente es un objetivo tan deseable que se hacen esfuerzos importantes para traducir los programas a estructuras ejecutables con eficiencia, en especial código de máquina interpretable por hardware. El proceso de traducción se vuelve gradualmente más complejo a medida que la forma ejecutable del programa se aleja más en cuanto a estructura respecto al programa original. En el extremo, un compilador optimizador para un lenguaje complejo como Ada puede alterar de manera radical las estructuras del programa para obtener una ejecución más eficiente. Estos compiladores se cuentan entre los programas más complejos que existen.

Desde el punto de vista lógico, la traducción se puede dividir en dos partes principales: el *análisis* del programa fuente de entrada y la *síntesis* del programa objeto ejecutable. Dentro de cada una de estas partes existen divisiones adicionales, como se verá en seguida. En casi todos los traductores, estas etapas lógicas no están claramente separadas, sino que se mezclan

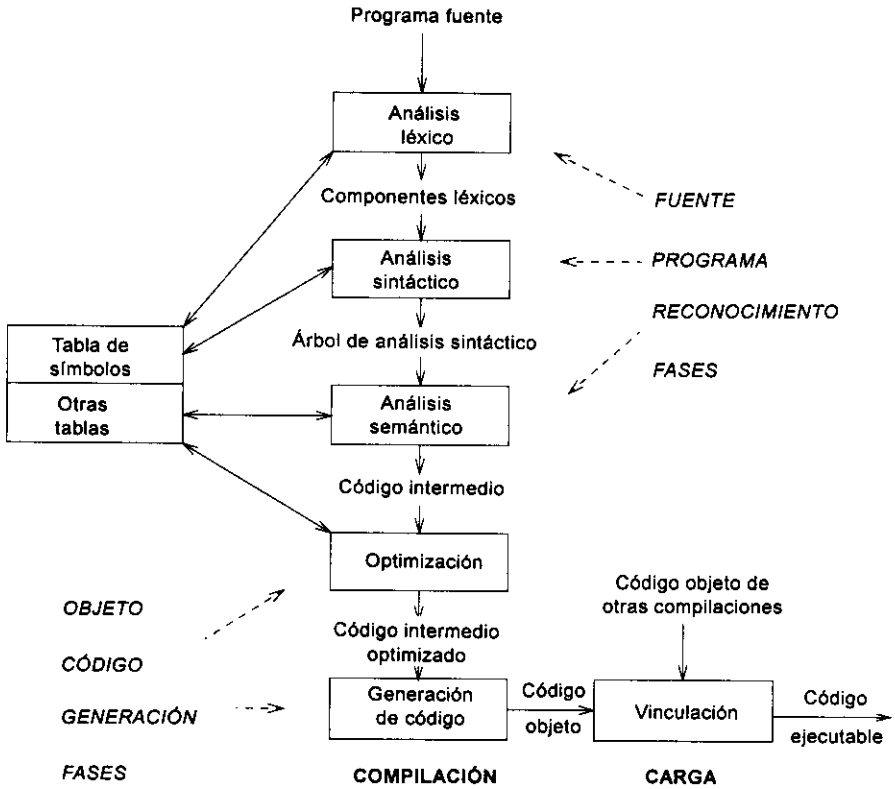


Figura 3.2. Estructura de un compilador.

de manera que se alterna el análisis con la síntesis, a menudo sobre la base de enunciado por enunciado. La figura 3.2 ilustra la estructura de un compilador típico.

Los traductores se agrupan en forma burda de acuerdo con el número de *pasos* que efectúan sobre el programa fuente. El compilador *estándar* (si es que se puede usar ese término) emplea típicamente dos pasos sobre el programa fuente. El primer paso de análisis descompone el programa en los componentes que lo constituyen y obtiene información, como el uso de nombres de variable, del programa. El segundo paso genera típicamente un programa objeto a partir de esta información recogida.

Si la velocidad de compilación es importante (como en un compilador educativo), se puede emplear una estrategia de un paso. En este caso, conforme el programa se analiza, se convierte de inmediato en código objeto. El Pascal fue proyectado de modo que se pudiera desarrollar un compilador de un paso para el lenguaje. Sin embargo, si la velocidad de ejecución tiene máxima importancia, se puede desarrollar un compilador de tres pasos (o más). El primer paso analiza el programa fuente, el segundo paso reescribe el programa fuente en una forma más eficiente usando diversos algoritmos de optimización bien definidos, y el tercer paso genera el código objeto.

Conforme la tecnología de compiladores ha mejorado, la relación entre el número de pasos y la velocidad del compilador ha perdido claridad. Lo que es más importante es la complejidad del lenguaje más que el número de pasos que se necesitan para analizar el programa fuente.

3.2.1 Análisis del programa fuente

Para un traductor, el programa fuente se presenta inicialmente como una serie larga y no diferenciada de símbolos, compuesta de miles o decenas de miles de caracteres. Desde luego, un programador que ve un programa así lo estructura casi de inmediato en subprogramas, enunciados, declaraciones, etc. Para el traductor nada de esto es manifiesto. Durante la traducción se debe construir laboriosamente, carácter por carácter, un análisis de la estructura del programa.

Análisis léxico. La fase fundamental de cualquier traducción es agrupar esta serie de caracteres en sus constituyentes elementales: identificadores, delimitadores, símbolos de operadores, números, palabras clave, palabras pregonadas, espacios en blanco, comentarios, etc. Esta fase se conoce como *análisis léxico*, y las unidades básicas de programa que resultan del análisis léxico se llaman *elementos* (o *componentes*) *léxicos*. Típicamente, el analizador léxico (o *revisor*) es la rutina de entrada para el traductor; lee renglones sucesivos del programa de entrada, los descompone en elementos léxicos individuales y alimenta estos elementos léxicos a las etapas posteriores del traductor para su uso en los niveles superiores de análisis. El analizador léxico debe identificar el tipo de cada elemento léxico (número, identificador, delimitador, operador, etc.) y adjuntar una marca de tipo. Además, se suele hacer la conversión a una representación interna de elementos como números (que se convierten a forma binaria interna de punto fijo o flotante) e identificadores (que se guardan en una tabla de símbolos y se usa la dirección de la entrada de la tabla de símbolos en lugar de la cadena de caracteres). El modelo básico que se usa para proyectar analizadores léxicos es el *autómata de estados finitos*, el cual se describe en forma breve en la sección 3.3.2.

Si bien el concepto de análisis léxico es sencillo, esta fase de la traducción suele requerir una mayor proporción del tiempo de traducción que cualquier otra. Este hecho se debe en parte simplemente a la necesidad de explorar y analizar el programa fuente carácter por carácter. También es cierto que en la práctica a veces es difícil determinar dónde se encuentran los límites entre elementos léxicos sin algoritmos bastante complejos y dependientes del contexto. Por ejemplo, los dos enunciados en FORTRAN:

$$\text{DO 10 I = 1,5} \quad \text{y} \quad \text{DO 10 I = 1.5}$$

tienen estructuras léxicas completamente distintas. El primero es un enunciado DO y el segundo es una asignación, pero este hecho no se puede descubrir sino hasta que se lee el carácter ya sea "," o ".", puesto que FORTRAN no toma en cuenta los espacios en blanco.

Análisis sintáctico. La segunda etapa de la traducción es el *análisis sintáctico* (*parsing*). En ella se identifican las estructuras de programa más grandes (enunciados, declaraciones, expresiones, etc.) usando los elementos léxicos producidos por el analizador léxico. El análisis sintáctico se alterna ordinariamente con el análisis semántico. Primero, el analizador sintáctico

identifica una serie de elementos léxicos que forman una unidad sintáctica como una expresión, enunciado, llamada de subprograma o declaración. Se llama entonces a un analizador semántico para que procese esta unidad. Por lo común, el analizador sintáctico y el semántico se comunican usando una pila. El analizador sintáctico introduce en la pila los diversos elementos de la unidad sintáctica hallada, y el analizador semántico los recupera y los procesa. Gran cantidad de investigación se ha enfocado al descubrimiento de técnicas eficientes de análisis sintáctico, en particular técnicas basadas en el uso de gramáticas formales (como se describe en la sección 3.3.1). En la sección 3.3.4 se estudia el análisis sintáctico con más detalle.

Análisis semántico. El análisis semántico es tal vez la fase medular de la traducción. Aquí, se procesan las estructuras sintácticas reconocidas por el analizador sintáctico y la estructura del código objeto ejecutable comienza a tomar forma. El análisis semántico es, por tanto, el puente entre las partes de análisis y de síntesis de la traducción. En esta etapa también ocurre un cierto número de otras funciones subsidiarias importantes, entre ellas el mantenimiento de tablas de símbolos, la mayor parte de la detección de errores, la expansión de macros y la ejecución de enunciados de tiempo de compilación. El analizador semántico puede producir en efecto el código objeto ejecutable en traducciones sencillas, pero es más común que la salida de esta etapa sea alguna forma interna del programa ejecutable final, la cual es manipulada luego por la etapa de optimización del traductor antes que se genere efectivamente código ejecutable.

El analizador semántico se divide ordinariamente en un conjunto de analizadores semánticos más pequeños, cada uno de los cuales maneja un tipo particular de construcción de programa. Los analizadores semánticos interactúan entre ellos mismos a través de información que se guarda en diversas estructuras de datos, en particular en la tabla central de símbolos. Por ejemplo, un analizador semántico que procesa declaraciones de tipo para variables sencillas suele poder hacer poco más que introducir los tipos declarados en la tabla de símbolos. Un analizador semántico posterior que procesa expresiones aritméticas puede usar luego los tipos declarados para generar las operaciones aritméticas apropiadas específicas de tipo para el código objeto. Las funciones exactas de los analizadores semánticos varían considerablemente, según el lenguaje y la organización lógica del traductor. Algunas de las funciones más comunes se pueden describir como sigue:

1. *Mantenimiento de tablas de símbolos.* Una *tabla de símbolos* es una de las estructuras de datos medulares de todo traductor. La tabla de símbolos contiene típicamente una entrada por cada identificador diferente encontrado en el programa fuente. El analizador léxico efectúa las introducciones iniciales conforme explora el programa de entrada, pero los analizadores semánticos tienen la responsabilidad principal a partir de ese momento. La entrada de tabla de símbolos contiene más que sólo el identificador mismo; contiene datos adicionales respecto a los atributos de ese identificador: su tipo (variable simple, nombre de arreglo, nombre de subprograma, parámetro formal, etc.), tipo de valores (enteros, reales, etc.), entorno de referimiento, y cualquier otra información disponible a partir del programa de entrada a través de declaraciones y uso. Los analizadores semánticos introducen esta información en la tabla de símbolos conforme procesan declaraciones, encabezados de programa y enunciados de programa. Otras partes del traductor usan esta información para construir código ejecutable eficiente.

La tabla de símbolos de los traductores para lenguajes compilados se suele desechar al final de la traducción. Sin embargo, puede retenerse durante la ejecución, por ejemplo, en lenguajes que permiten crear nuevos identificadores durante la ejecución o como ayuda para la depuración. Todas las implementaciones de ML, Prolog y LISP utilizan una tabla de símbolos creada inicialmente durante la traducción como una estructura de datos central definida por el sistema en tiempo de ejecución. *dbx* es un popular programa de UNIX que utiliza una tabla de símbolos en tiempo de ejecución para depurar programas en C.

2. *Inserción de información implícita.* A menudo, en los programas fuente, la información está implícita y debe hacerse explícita en los programas objeto de nivel más bajo. La mayor parte de esta información implícita se sitúa abajo del encabezado de *convenciones predeterminadas*, interpretaciones que se proveen cuando el programador proporciona una especificación no explícita. Por ejemplo, una variable FORTRAND que se usa pero no se declara se provee automáticamente de una declaración tipo que depende de la inicial de su nombre.

3. *Detección de errores.* Los analizadores sintácticos y semánticos deben estar preparados para manejar programas tanto incorrectos como correctos. En cualquier punto, el analizador léxico puede enviar al analizador sintáctico un elemento sintáctico que no encaja en el contexto circundante (por ejemplo, un delimitador de enunciado en medio de una expresión, una declaración a la mitad de una serie de enunciados, un símbolo de operador donde se espera un identificador). El error puede ser más sutil, como una variable real donde se requiere una variable entera o una referencia de variable de subíndice con tres subíndices cuando se declaró que el arreglo tenía dos dimensiones. En cada paso de la traducción puede ocurrir una multitud de errores de este tipo. El analizador semántico no sólo debe reconocer estos errores cuando se presentan y generar un mensaje de error apropiado, sino también, en todos los casos excepto en los más drásticos, determinar la manera apropiada de continuar con el análisis sintáctico del resto del programa. Las medidas para detección y manejo de errores en las fases de análisis sintáctico y semántico pueden requerir un esfuerzo mayor que el análisis básico mismo.

4. *Procesamiento de macros y operaciones en tiempo de compilación.* No todos los lenguajes incluyen capacidades para macros o recursos para operaciones en tiempo de compilación. Cuando están presentes, sin embargo, el procesamiento se maneja comúnmente durante el análisis semántico.

   Una *macro*, en su forma más sencilla, es un trozo de texto de programa que se ha definido por separado y que se va a insertar en el programa durante la traducción siempre que se encuentre una *llamada de macro* en el programa fuente. Así pues, una macro se parece mucho a un subprograma, excepto que en vez de traducirla y llamarla durante la ejecución (es decir, el enlace del nombre del subprograma con su semántica tiene lugar en tiempo de ejecución), su cuerpo se sustituye simplemente por cada llamada durante la traducción del programa (es decir, el enlace ocurre en tiempo de traducción). Las

macros pueden ser sólo simples cadenas que se van a sustituir, por ejemplo, la sustitución de 3.1416 por PI siempre que se hace referencia a PI. Es más común que se parezcan mucho a los subprogramas, con parámetros que se deben procesar antes que se haga la sustitución de la llamada de macro.

Cuando se permiten macros, los analizadores semánticos deben identificar la llamada de macro dentro del programa fuente y establecer la sustitución apropiada de la llamada por el cuerpo de la macro. Esta tarea suele hacer necesario interrumpir a los analizadores léxico y sintáctico y ponerlos a trabajar analizando la cadena que representa el cuerpo de la macro antes de continuar con el resto de la cadena fuente. Alternativamente, el cuerpo de la macro puede haber sido ya parcialmente traducido, de modo que el analizador semántico puede procesarlo directamente insertando el código objeto apropiado y asentando las entradas de tabla apropiadas antes de continuar con el análisis del programa fuente.

Una *operación en tiempo de compilación* es una operación que se va a llevar a cabo durante la traducción para controlar la traducción del programa fuente. C suministra un cierto número de operaciones de esta clase. La "#define" de C permite evaluar las constantes o expresiones antes de compilar el programa. La construcción "#ifdef" (si se define) permite compilar secuencias alternativas de código de acuerdo con la presencia o ausencia de ciertas variables. Estos conmutadores permiten al programador alterar el orden de los enunciados que se compilan. Por ejemplo, un archivo fuente común se puede usar para compilar versiones alternativas de un programa, como se muestra:

```
#define pc                  /* Fijar a versión PC o UNIX del programa */
   ...
ProgramWrite(...)
   ...
#ifdef pc                   /* Si se define entonces se necesita código PC */
                            /* Hacer código de versión PC */
                            /* p. ej. escribir salida Windows Microsoft */
#else
   ...                      /* Hacer código de versión UNIX */
                            /* p. ej. escribir salida Windows Motif X */
#endif
```

3.2.2 Síntesis del programa objeto

Las etapas finales de la traducción se ocupan de la construcción del programa ejecutable a partir de las salidas que produce el analizador semántico. Esta fase implica necesariamente generación de código y también puede incluir optimización del programa generado. Si los subprogramas se traducen por separado, o si se emplean subprogramas de biblioteca, se necesita una etapa final de vinculación y carga para producir el programa completo listo para ejecutarse.

Optimización. El analizador semántico produce ordinariamente como salida el programa ejecutable traducido representado en algún *código intermedio*, una representación interna como una cadena de operadores y operandos o una tabla de series de operador/operando. A partir de esta representación interna, los generadores de código pueden crear el código objeto de salida con el formato apropiado. Sin embargo, antes de la generación de código, se lleva a cabo ordinariamente cierta optimización del programa en la representación interna. Típicamente, el analizador semántico genera la forma del programa interno de manera irregular, conforme se analiza cada segmento del programa de entrada. El analizador semántico no tiene que preocuparse en general acerca del código circundante que ya se ha generado. Al elaborar esta salida poco sistemática, sin embargo, se puede producir código extremadamente deficiente; por ejemplo, un registro interno se puede guardar al final de un segmento generado y volverse a cargar de inmediato, a partir de la misma localidad, al principio del próximo segmento. Por ejemplo, el enunciado:

$$A = B + C + D$$

puede generar el código intermedio:

(a) Temp1 = B + C
(b) Temp2 = Temp1 + D
(c) A = Temp2

el cual puede generar el código sencillo aunque ineficiente:

1. Cargar registro con B (a partir de (a))
2. Sumar C al registro
3. Guardar registro en Temp1
4. Cargar registro con Temp1 (a partir de (b))
5. Sumar D al registro
6. Guardar registro en Temp2
7. Cargar registro con Temp2 (a partir de (c))
8. Guardar registro en A

Las instrucciones 3 y 4, así como las 6 y 7, son redundantes, puesto que todos los datos se pueden conservar en el registro antes de guardar el resultado en *A*. Suele ser deseable permitir que los analizadores semánticos generen secuencias de código deficientes y luego, durante la optimización, reemplazar estas secuencias por otras mejores que evitan ineficiencias obvias.

Muchos compiladores van mucho más allá de esta clase de optimización simple y analizan el programa en busca de otras mejoras susceptibles de llevarse a cabo, por ejemplo, cómputo de subexpresiones comunes una sola vez, eliminación de operaciones con constantes de las iteraciones, optimización del uso de registros internos y del cálculo de fórmulas de acceso a arreglos. Es mucho lo que se ha investigado en cuanto a optimización de programas y se conocen muchas técnicas refinadas (véanse las referencias al final de este capítulo).

Generación de código. Después que se ha optimizado el programa traducido en la representación interna, se debe transformar en los enunciados en lenguaje ensamblador, código de máquina u otra forma de programa objeto que va a constituir la salida de la traducción. Este proceso implica dar el formato apropiado a la salida con base en la información que contiene la representación interna del programa. El código de salida puede ser directamente ejecutable o puede haber otros pasos de traducción por seguir, por ejemplo, ensamblado o vinculación y carga.

Vinculación y carga. En la etapa final optativa de la traducción, los fragmentos de código que son resultado de las traducciones individuales de subprogramas se funden en el programa final ejecutable. La salida de las fases de traducción precedentes consiste típicamente en programas ejecutables en una forma casi final, excepto cuando los programas hacen referencia a datos externos u otros subprogramas. Estas ubicaciones incompletas en el código se especifican en las *tablas de cargador* anexas que produce el traductor. El *cargador vinculador* (o *editor de vínculos*) carga los diversos segmentos de código traducido en la memoria y luego usa las tablas de cargador anexas para vincularlos correctamente entre sí introduciendo datos y direcciones de subprograma en el código según se requiere. El resultado es el programa ejecutable final listo para usarse.

Compiladores de diagnóstico

Una variante de este enfoque era popular durante los años sesenta, cuando las computadoras de procesamiento por lotes causaban grandes retrasos en la recepción de los resultados de una compilación. La sinopsis de lenguaje 3.2 sintetiza la experiencia de la Universidad Cornell en el desarrollo de compiladores de diagnóstico para CORC [CONWAY y MAXWELL 1963], CUPL, y PL/C [CONWAY y WILCOX 1973]. En este caso, el procesamiento rápido y tiempo corto de compilación era la meta principal, puesto que los programas estudiantiles se ejecutaban típicamente durante sólo una fracción de segundo y, una vez ejecutados correctamente, se desechaban.

3.3 MODELOS FORMALES DE TRADUCCIÓN

Como se expresó en la sección anterior, las partes de reconocimiento sintáctico de la teoría de compiladores son bastante normales y se basan en general en la teoría de lenguajes independiente del contexto. En las páginas siguientes se presenta un breve resumen de esa teoría.

La definición formal de la sintaxis de un lenguaje de programación se conoce ordinariamente como una *gramática*, en analogía con la terminología común para los lenguajes naturales. Una gramática se compone de un conjunto de *reglas* (llamadas *producciones*) que especifican las series de caracteres (o elementos léxicos) que forman programas permisibles en el lenguaje que se está definiendo. Una *gramática formal* es simplemente una gramática que se especifica usando una notación definida de manera estricta. Las dos clases de gramáticas útiles en tecnología de compiladores incluyen la *gramática BNF* (o *gramática libre del contexto*) y la *gramática normal*, las cuales se describen en las secciones siguientes. En la sección 9.3 se ofrece un breve resumen de otras clases de gramáticas, que no son particu-

Sinopsis de lenguaje 3.2: CORC, CUPL y PL/C

Características: Se suministran compiladores para corrección automática de programas inválidos.

Historia: Durante la era del procesamiento por lotes de los años sesenta y hasta mediados de los setenta, solía tomar más de un día la devolución de un programa compilado. Los errores de compilación eran por tanto muy costosos. La Universidad Cornell, bajo el liderazgo de Richard Conway, desarrolló una serie de compiladores (CORC, Cornell Compiler; CUPL, Cornell University Programming Language; PL/C, dialecto de Cornell para PL/I) que efectuaban automáticamente correcciones de errores para reparar errores sintácticos y semánticos simples como ayuda para hacer más rápida la ejecución de los programas. Para errores sencillos, estos sistemas eran bastante eficaces. Con el crecimiento de sistemas de tiempo compartido en los años setenta y la computadora personal y la estación de trabajo de los ochenta, la necesidad de corrección automática de errores ha decaído.

Ejemplo:

```
/*EJEMPLO EN PL/C DE CORRECCIÓN DE ERRORES EN PL/I */
SAMPLE: PROCEDURE OPTIONS(MAIN);   /*PROCEDIMIENTO PRINCIPAL */
DECLARE VAR1 FIXED BINARY /* VAR1 entero; Corregir ; faltante al final */
   DECLARE VAR2 FIXED BINARY;
   VAR1 = 1;
   VAR2 = 2;
   IF (VAR1 < VAR2 /*Agregar ) faltante al final de renglón */
      THEN PUT SKIP LIST(VAR1, 'menor que', VAR2) /* Agregar ; al final */
         /* Imprimir: 1 menor que 2 */
         /* SKIP pasa al nuevo renglón */
      ELSE PUT SKIP LIST(VAR2, 'menor que', VAR1);
   DO WHILE(VAR2>0);
      PUT SKIP ('Cuenta regresiva', VAR2);   /* Agregar término LIST faltante */
      VAR2 VAR2 - 1;   /* = insertado después de primera VAR2 */
      END;        /* END DO WHILE */
   END;           /* END PROCEDURE SAMPLE */
```

Referencia: R. W. Conway y T. Wilcox, "Design and Implementation of a Diagnostic Compiler for PL/I," *Comm. ACM* (16)3 (1973), 169-179.

larmente útiles para el desarrollo de compiladores pero tienen una importancia extrema para entender las capacidades generales de cálculo que proporcionan las computadoras.

3.3.1 Gramáticas BNF

Cuando se considera la estructura de una oración en español, se le describe por lo general como una secuencia de categorías. Es decir, una oración sencilla se suele dar como:

sujeto / verbo / complemento

de lo cual son ejemplos:

La niña / corrió / a casa.
El muchacho / prepara / la comida.

Cada categoría se puede dividir aún más. En los ejemplos anteriores, el *sujeto* está representado por *artículo y nombre*, por lo que la estructura de estas oraciones es:

artículo / nombre / verbo / complemento

Existen otras estructuras de oración posibles además de las declarativas simples que se han citado. Las oraciones interrogativas (preguntas) simples suelen tener esta sintaxis:

verbo auxiliar / sujeto / predicado

como en:

¿Estaba / la niña / corriendo a casa?
¿Está / el muchacho / preparando la comida?

Podemos representar estas oraciones por un conjunto de reglas. Podemos decir que una oración puede ser una oración declarativa simple o una oración interrogativa simple, o, de manera notativa, como:

⟨*oración*⟩ ::= ⟨*declarativa*⟩ | ⟨*interrogativa*⟩

donde ::= significa "se define como" y | significa "o". Cada tipo de oración se puede definir adicionalmente como:

⟨*declarativa*⟩ ::=	⟨*sujeto*⟩ ⟨*verbo*⟩ ⟨*complemento*⟩.
⟨*sujeto*⟩ ::=	⟨*artículo*⟩ ⟨*nombre*⟩
⟨*interrogativa*⟩ ::=	¿⟨*verbo auxiliar*⟩ ⟨*sujeto*⟩ ⟨*predicado*⟩?

Esta notación específica se conoce como *BNF* (*Backus Naur form*; forma de Backus Naur) y fue desarrollada para la definición sintáctica de ALGOL por John Backus [BACKUS 1960] a finales de la década de 1950 como una forma de expresar estas ideas para lenguajes de programación. Peter Naur era presidente del comité que desarrolló ALGOL. Al mismo tiempo, el lingüista Noam Chomsky desarrolló una forma gramatical similar, la *gramática libre del contexto* [CHOMSKY 1959] para la definición de la sintaxis de lenguajes naturales. La BNF y la gramática libre del contexto son equivalentes en cuanto a poder; las diferencias corresponden sólo a la notación. Por esta razón, los términos *gramática BNF* y *gramática libre del contexto* son ordinariamente intercambiables en el estudio de la sintaxis.

Sintaxis

Una gramática BNF se compone de un conjunto finito de reglas de gramática BNF, las cuales definen un *lenguaje*, que en nuestro caso es un lenguaje de programación. Antes de examinar estas reglas gramaticales en detalle, el término *lenguaje* amerita aquí cierta explicación adicional. Puesto que la sintaxis se ocupa sólo de la forma y no del significado, un lenguaje (de programación), considerado desde el punto de vista sintáctico, se compone de un conjunto de *programas sintácticamente correctos*, cada uno de los cuales es simplemente una serie de caracteres. Un programa sintácticamente correcto no necesita tener sentido

semánticamente; por ejemplo, si se ejecutara, no tendría que computar algo útil, o nada en absoluto, para el caso.

Por ejemplo, considerando nuestras oraciones declarativas e interrogativas simples precedentes, la sintaxis *sujeto verbo complemento* también se satisface con la secuencia:

La casa /corrió / a niña

que no tiene sentido con las interpretaciones normales de estas palabras.

En el caso de la sintaxis de lenguajes de programación, esta falta de preocupación por el significado se lleva un paso más adelante: Un *lenguaje* es *cualquier conjunto de cadenas de caracteres* (de longitud finita) con caracteres elegidos de algún alfabeto finito fijo de símbolos. Bajo esta definición, todos los siguientes son lenguajes:

1. El conjunto de todos los enunciados de asignación en C
2. El conjunto de todos los programas en C
3. El conjunto de todos los átomos en LISP
4. El conjunto compuesto de secuencias de letras *a* y *b* donde todas las *a* anteceden a todas las *b* (por ejemplo, *ab*, *aab*, *abb*, ...)

Un lenguaje puede consistir en sólo un conjunto finito de cadenas (por ejemplo, el lenguaje compuesto de todos los delimitadores de Pascal: **begin**, **end**, **if**, **then**, etc.), o puede contener un número infinito de cadenas (por ejemplo, la cadena de letras *a* y *b* dada como número 4 en la lista anterior). La única restricción a un lenguaje es que cada cadena que contenga debe ser de longitud finita e incluir caracteres elegidos de algún alfabeto finito fijo de símbolos.

El estudio de la lista anterior de lenguajes de muestra indica algunos de los problemas en el uso del lenguaje natural (español en este caso) para describir lenguajes de programación. Considérese una vez más el número 4. ¿Es *b* por sí misma un miembro de este lenguaje? Es verdad que todas las *a* (ninguna en este caso) anteceden a todas las *b* de la cadena, pero, ¿debe una cadena contener al menos una *a*? De manera similar, ¿está *a* por sí misma en el lenguaje? Tal como se ha expresado, la descripción es incompleta.

Este problema se resuelve proporcionando un conjunto matemático formal de reglas para determinar con exactitud cuáles cadenas están en el lenguaje. En el caso más sencillo, una regla gramatical puede simplemente enumerar los elementos de un lenguaje finito. Por ejemplo:

⟨*dígito*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Esta regla define un lenguaje compuesto de las 10 cadenas de un solo carácter 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9 enumerando un conjunto de alternativas. La regla gramatical precedente se lee como "Un *dígito* es ya sea un '0' o un '1' o un '2' o un ...". El término *dígito* se conoce como una *categoría sintáctica* o un *no terminal*; sirve como nombre para el lenguaje que define la regla gramatical. Los símbolos que componen cadenas en nuestro lenguaje, en este caso los dígitos del 0 al 9, se llaman símbolos *terminales*. El símbolo ::= se suele dar como →, en especial si las categorías sintácticas se escriben como letras mayúsculas solas

(por ejemplo, la regla ⟨*X*⟩ ::= ⟨*B*⟩ | ⟨*C*⟩ se suele escribir como $X \rightarrow B|C$). En este libro se usan ambas notaciones.

Una vez que se ha definido un conjunto básico de categorías sintácticas, se pueden usar para construir cadenas más complejas. Por ejemplo, la regla:

⟨*enunciado condicional*⟩ ::=
if ⟨*expresión booleana*⟩ **then** ⟨*enunciado*⟩
else ⟨*enunciado*⟩ |
if ⟨*expresión booleana*⟩ **then** ⟨*enunciado*⟩

define el lenguaje compuesto de construcciones de ⟨*enunciado condicional*⟩ y que emplea las categorías sintácticas ⟨*expresión booleana*⟩ y ⟨*enunciado*⟩, las cuales se deben definir a su vez usando otras reglas gramaticales. Obsérvese que la regla anterior muestra dos formas alternativas de enunciado condicional (separadas por el símbolo |). Cada alternativa se construye a partir de la concatenación de varios elementos, que pueden ser cadenas literales (por ejemplo, **if** o **else**) o categorías sintácticas. Cuando se designa una categoría sintáctica, significa que en ese punto se puede usar cualquier cadena del sublenguaje definida por esa categoría.. Por ejemplo, si se supone que *expresión booleana* consiste en un conjunto de cadenas que representan expresiones booleanas válidas, la regla anterior permite insertar cualquiera de estas cadenas entre el **if** y el **then** de un enunciado condicional.

La categoría sintáctica que define la regla puede usarse ella misma en esa regla para especificar repetición. Esta clase de regla se conoce como regla *recursiva*. Por ejemplo, la regla recursiva:

⟨*entero sin signo*⟩ ::= ⟨*dígito*⟩ | ⟨*entero sin signo*⟩⟨*dígito*⟩

define un entero sin signo como una serie de ⟨*dígito*⟩s. La primera alternativa de ⟨*dígito*⟩ define un entero sin signo como un solo dígito, mientras que la segunda alternativa agrega un segundo dígito a este dígito inicial, un tercer dígito al segundo, y así sucesivamente.

Una gramática BNF completa es tan solo un conjunto de esta clase de reglas gramaticales, las cuales definen en conjunto una jerarquía de sublenguajes que conduce a la categoría sintáctica de nivel máximo, la cual, para un lenguaje de programación, es por lo común la categoría sintáctica ⟨*programa*⟩. La figura 3.3 ilustra una gramática más compleja que define la sintaxis de una clase de enunciados de asignación simples, la cual supone que ya se han definido las categorías sintácticas básicas ⟨*identificador*⟩ y ⟨*número*⟩.

Árboles de análisis sintáctico

Dada una gramática, podemos usar una regla de reemplazo único para generar cadenas en nuestro lenguaje. Por ejemplo, la gramática siguiente genera todas las series de paréntesis equilibrados:

$$S \rightarrow SS \mid (S) \mid ( )$$

⟨*enunciado de asignación*⟩ ::= ⟨*variable*⟩ = ⟨*expresión aritmética*⟩
⟨*expresión aritmética*⟩ ::= ⟨*término*⟩ | ⟨*expresión aritmética*⟩ + ⟨*término*⟩ | ⟨*expresión aritmética*⟩ – ⟨*término*⟩
⟨*término*⟩ ::= ⟨*primario*⟩ | ⟨*término*⟩ × ⟨*primario*⟩ | ⟨*término*⟩/⟨*primario*⟩
⟨*primario*⟩ ::= ⟨*variable*⟩ | ⟨*número*⟩ | (⟨*expresión aritmética*⟩)
⟨*variable*⟩ ::= ⟨*identificador*⟩ | ⟨*identificador*⟩[⟨*lista de subíndices*⟩]
⟨*lista de subíndices*⟩ ::= ⟨*expresión aritmética*⟩ | ⟨*lista de subíndices*⟩, ⟨*expresión aritmética*⟩

Figura 3.3. Gramática para enunciados de asignación simples.

Conocida cualquier cadena, se puede reemplazar cualquier no terminal por el lado derecho de cualquier producción que contenga ese no terminal a la izquierda. Por ejemplo, se puede generar la cadena (( )( )) a partir de *S*:

Reemplazando *S* usando la regla $S \rightarrow (S)$ para dar (S)
Reemplazando *S* en (S) usando la regla $S \rightarrow SS$ para dar (SS)
Reemplazando primero *S* en (SS) usando la regla $S \rightarrow ( )$ para dar $(( )S)$
Reemplazando *S* en $(( )S)$ usando la regla $S \rightarrow ( )$ para dar $(( )( ))$

Se usará el símbolo $\Rightarrow$ para indicar que una cadena se *deriva* de otra cadena. La derivación completa se puede escribir como:

$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (( )S) \Rightarrow (( )( ))$$

Cada término de la derivación se conoce como una *forma sentencial*, y definimos formalmente un lenguaje como *el conjunto de formas sentenciales, cada una compuesta sólo de símbolos terminales*, que se puede derivar del símbolo inicial de una gramática.

El uso de una gramática formal para definir la sintaxis de un lenguaje de programación es importante tanto para el usuario del lenguaje como para el implementador del mismo. El usuario puede consultarla para responder preguntas sutiles acerca de forma, puntuación y estructura del programa. El implementador puede usarla para determinar todos los casos posibles de estructuras de programa de entrada que se permiten y, en esta forma, con cuál de ellas puede tener que vérselas el traductor. Y tanto el programador como el implementador tienen una definición común acordada que se puede usar para dirimir disputas acerca de construcciones sintácticas permisibles. Una definición sintáctica formal también ayuda a eliminar diferencias sintácticas menores entre implementaciones de un lenguaje.

Para determinar si una cadena dada representa de hecho un programa sintácticamente válido en el lenguaje definido por una gramática BNF, se deben usar las reglas gramaticales para construir un análisis sintáctico de la cadena. Si la cadena se puede analizar sintácticamente en forma satisfactoria, entonces está en el lenguaje. Si no se puede encontrar alguna forma de analizar sintácticamente la cadena con las reglas gramaticales dadas, entonces la cadena no

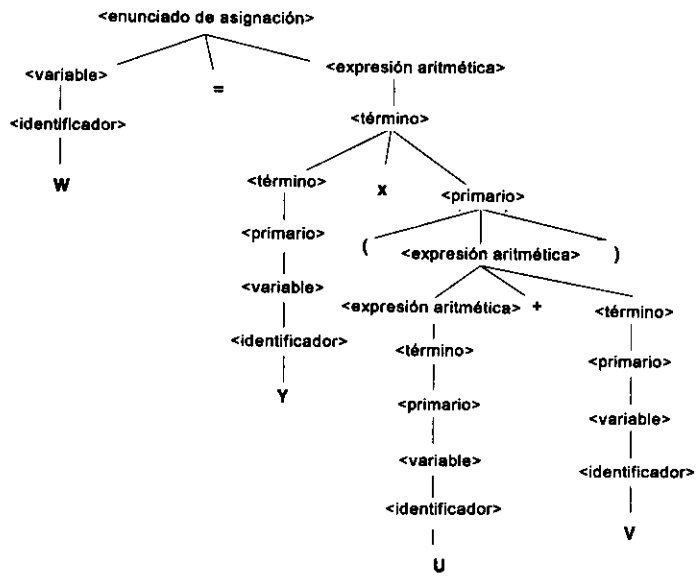


Figura 3.4. Árbol de análisis sintáctico para un enunciado de asignación.

está en el lenguaje. La figura 3.4 ilustra el *árbol de análisis sintáctico* que es el resultado de un análisis sintáctico del enunciado $W = Y + (U + V)$ usando la gramática BNF de la figura 3.3.

Una gramática BNF asigna una estructura a cada cadena que está en el lenguaje definido por la gramática, como se ve en la figura 3.4. Obsérvese que la estructura asignada es necesariamente un árbol a causa de las restricciones a las reglas de la gramática BNF. Cada hoja de este árbol de análisis sintáctico es un solo carácter o elemento léxico de la cadena de entrada. Cada punto intermedio de bifurcación del árbol está marcado con una categoría sintáctica que designa la clase a la cual pertenece el subárbol que está abajo de él. El nodo raíz del árbol está marcado con la categoría sintáctica que designa al lenguaje completo, en este caso la categoría ⟨*enunciado de asignación*⟩.

El árbol de análisis sintáctico suministra una estructura semántica intuitiva para gran parte de un programa. Así, por ejemplo, la gramática BNF para Pascal especifica una estructura para un programa como una serie de declaraciones y enunciados, con bloques anidados. Los enunciados, a su vez, se estructuran usando expresiones de diversas clases, y las expresiones se componen de variables simples y subindizadas, operadores de primitivas, llamadas de funciones, etc. Al nivel más bajo, incluso los identificadores y números se descomponen en las partes que los constituyen. A través del estudio de la gramática, el programador puede alcanzar una comprensión directa de diversas estructuras que se combinan para formar programas correctos. Es importante señalar que ninguna gramática debe asignar *necesariamente* la estructura que uno esperaría a un elemento de programa dado. El mismo lenguaje puede estar definido por muchas gramáticas distintas, como se puede ver fácilmente si se juega con la gramática de la figura 3.3. La figura 3.5, por ejemplo, presenta una gramática que define el

⟨*enunciado de asignación*⟩ ::= ⟨*variable*⟩ = ⟨*expresión aritmética*⟩
⟨*expresión aritmética*⟩ ::= ⟨*término*⟩ | ⟨*expresión aritmética*⟩ × ⟨*término*⟩ | ⟨*expresión aritmética*⟩ + ⟨*término*⟩
⟨*término*⟩ ::= ⟨*primario*⟩ | ⟨*término*⟩ − ⟨*primario*⟩ | ⟨*término*⟩/⟨*primario*⟩
⟨*primario*⟩ ::= ⟨*variable*⟩ | ⟨*número*⟩ | (⟨*expresión aritmética*⟩)
⟨*variable*⟩ ::= ⟨*identificador*⟩ | ⟨*identificador*⟩[⟨*lista de subíndices*⟩]
⟨*lista de subíndices*⟩ ::= ⟨*expresión aritmética*⟩ | ⟨*lista de subíndices*⟩, ⟨*expresión aritmética*⟩

Figura 3.5. Gramática BNF alternativa.

mismo lenguaje que la gramática de la figura 3.3, pero adviértase que las estructuras que asigna esta nueva gramática están bastante en desacuerdo con las estructuras que uno asignaría intuitivamente.

La gramática BNF, a pesar de su estructura en extremo simple, se puede usar para hacer un trabajo sorprendentemente bueno al definir la sintaxis de casi todos los lenguajes de programación. Las áreas de sintaxis que no pueden definirse con una gramática BNF son aquellas que implican dependencia contextual. Por ejemplo, las restricciones "el mismo identificador no se puede declarar dos veces en el mismo bloque", "todo identificador se debe declarar en algún bloque que encierre el punto de su uso", y "un arreglo para el que se ha declarado que tiene dos dimensiones no puede ser referido con tres subíndices" son todas no especificables usando sólo una gramática BNF. Las restricciones de este tipo se deben definir por medio de un apéndice a la gramática BNF formal.

El proceso por el cual se emplea una gramática BNF para desarrollar árboles de análisis sintáctico para un programa dado es un proceso que se comprende bien. En la sección 3.3.4 se describe en forma breve una estrategia sencilla de análisis sintáctico, el descenso recursivo, para suministrar una apreciación de los problemas que esto implica.

Ambigüedad

Como ya se ha señalado, la ambigüedad es un problema que tiene que ver con la sintaxis. Considérese la expresión "nada en el agua". Se puede representar de dos maneras:

Nada / en el agua
/ Nada en el agua

Ambas tienen un significado propio, aunque muy diferente. En el primer caso, *Nada* es un pronombre que se refiere a la ausencia de cualquier cosa en el agua, mientras que, en el segundo caso, *Nada* es un verbo en forma imperativa que induce a alguien a nadar en el agua.

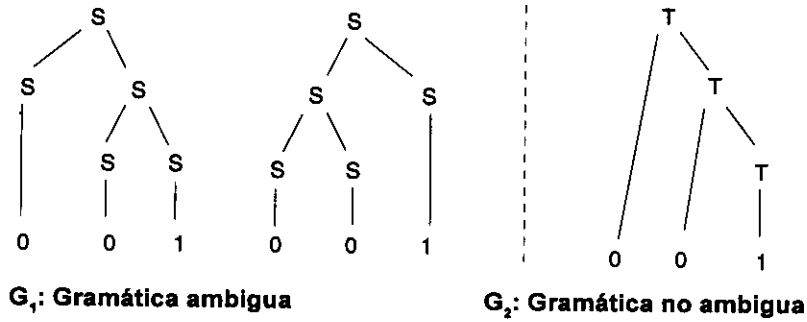


Figura 3.6. Ambigüedad en las gramáticas.

La ambigüedad suele ser una propiedad de una gramática dada y no de un lenguaje específico. Por ejemplo, la gramática G_1, que genera todas las cadenas binarias, es ambigua:

$$G_1 : S \rightarrow SS \mid 0 \mid 1$$

Sabemos que es ambigua, puesto que existe alguna cadena en el lenguaje que tiene dos árboles de análisis sintáctico distintos (figura 3.6).

Si toda gramática para un lenguaje dado es ambigua, se dice que el lenguaje es *inherentemente ambiguo*. Sin embargo, el lenguaje de todas las cadenas binarias no es inherentemente ambiguo porque existe una gramática no ambigua que genera las mismas cadenas, la gramática G_2:

$$G_2 : T \rightarrow 0T \mid 1T \mid 0 \mid 1$$

Extensiones a la notación BNF

No obstante el poder, elegancia y sencillez de las gramáticas BNF, no constituyen una notación ideal para comunicar las reglas de sintaxis de lenguajes de programación al programador práctico. La razón primordial es la sencillez de la regla de BNF, la cual obliga a una representación bastante poco natural de las construcciones sintácticas comunes de elementos optativos, elementos alternativos y elementos repetidos dentro de una regla gramatical. Por ejemplo, para expresar la idea sintáctica simple "un *entero con signo* es una serie de dígitos precedida de un signo opcional de más o menos" debemos escribir en BNF un conjunto bastante complejo de reglas recursivas tales como:

⟨*entero con signo*⟩ ::= + ⟨*entero*⟩ | – ⟨*entero*⟩
⟨*entero*⟩ ::= ⟨*dígito*⟩ | ⟨*entero*⟩ ⟨*dígito*⟩

Podemos describir extensiones a la BNF que evitan algunas de estas formas no naturales de especificar propiedades sintácticas simples de ciertas gramáticas.

⟨*enunciado de asignación*⟩ ::= ⟨*variable*⟩ = ⟨*expresión aritmética*⟩
⟨*expresión aritmética*⟩ ::= ⟨*término*⟩ {[+ | −] ⟨*término*⟩ }*
⟨*término*⟩ ::= ⟨*primario*⟩ {[× | /] ⟨*primario*⟩}*
⟨*primario*⟩ ::= ⟨*variable*⟩ | ⟨*número*⟩ | (⟨*expresión aritmética*⟩)
⟨*variable*⟩ ::= ⟨*identificador*⟩ | ⟨*identificador*⟩[⟨*lista de subíndices*⟩]
⟨*lista de subíndices*⟩ ::= ⟨*expresión aritmética*⟩ {, ⟨*expresión aritmética*⟩}*

Figura 3.7. BNF extendida para enunciados de asignación simples.

Notación BNF extendida. Para ampliar la BNF, las extensiones de notación siguientes no modifican el poder de la gramática BNF pero dan cabida a descripciones más fáciles de los lenguajes:

- Se puede indicar un elemento optativo encerrando el elemento entre paréntesis rectangulares, [...].
- Una selección de alternativas puede usar el símbolo | dentro de una sola regla, opcionalmente encerrado entre paréntesis ([,]) si es necesario.
- Una serie arbitraria de casos de un elemento se puede indicar encerrando el elemento entre llaves seguidas de un asterisco, {...}*.

Son ejemplos de esta notación:

enteros con signo: ⟨*entero con signo*⟩ ::= [+ | −] ⟨*dígito*⟩{⟨*dígito*⟩}*
identificador: ⟨*identificador*⟩ ::= ⟨*letra*⟩ {⟨*letra*⟩ | ⟨*dígito*⟩}*

Como un ejemplo más, las expresiones aritméticas, como las que se describen en la figura 3.3, se pueden describir de manera más intuitiva usando BNF extendida. Las reglas como:

⟨*expresión aritmética*⟩ ::= ⟨*término*⟩ | ⟨*expresión aritmética*⟩ + ⟨*término*⟩

reflejan la relación de que una expresión aritmética es en realidad un número arbitrario de términos y se puede expresar usando BNF extendida como:

⟨*expresión aritmética*⟩ ::= ⟨*término*⟩ { + ⟨*término*⟩}*

La expresión gramatical completa se puede replantear en BNF extendida como se muestra en la figura 3.7.

Diagramas de sintaxis. Un diagrama de sintaxis (también llamado *diagrama de ferrocarril* porque se parece al patio de maniobras de un ferrocarril) es una forma gráfica de expresar reglas de BNF extendida. Cada regla está representada por un camino que va de la entrada a la izquierda a la salida a la derecha. Cualquier trayecto válido de entrada a salida representa una cadena generada por esa regla. Si representamos las otras reglas con cuadros y los símbolos

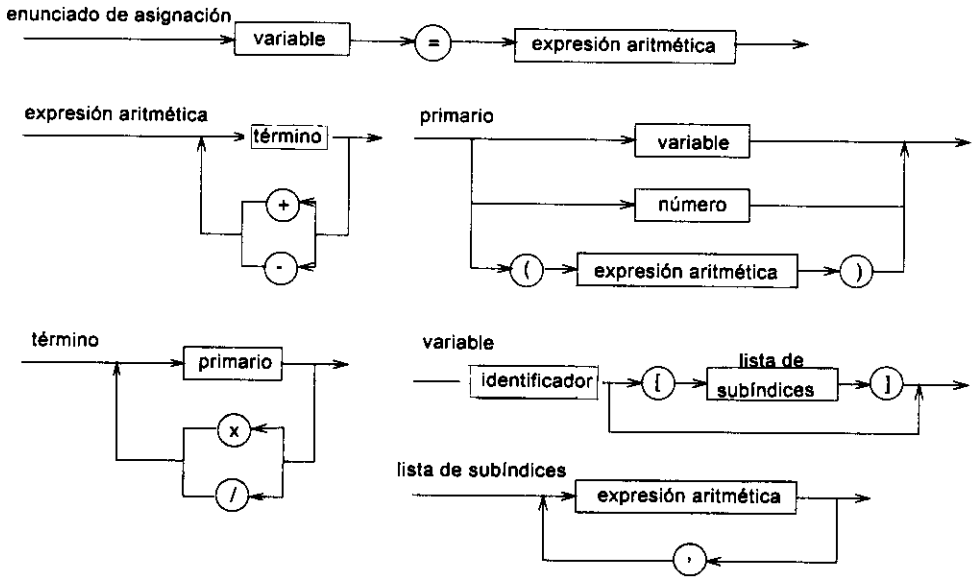


Figura 3.8. Diagramas de sintaxis para enunciados de asignación simples.

terminales con círculos, podemos representar la gramática de la figura 3.7 con los diagramas de sintaxis de la figura 3.8.

Por ejemplo, la producción ⟨*término*⟩ es satisfecha por cualquier camino que pase a través de ⟨*primario*⟩ y salga por la derecha, o pase a través de ⟨*primario*⟩, luego forme una iteración o más a través de uno de los operadores y después pase de nuevo a través de ⟨*primario*⟩, es decir, la regla de BNF extendida:

$$\langle término \rangle ::= \langle primario \rangle \{[\times \mid /] \langle primario \rangle\}^*$$

3.3.2 Autómatas de estados finitos

Todos los componentes léxicos de un lenguaje de programación tienen una estructura sencilla. Como ya se ha mencionado, la fase de análisis léxico del compilador descompone el programa fuente en un caudal de estos componentes léxicos. Adoptando aquí una perspectiva diferente en comparación con la sección anterior sobre gramáticas BNF, se describirá un modelo de máquina para reconocer componentes léxicos.

Un identificador comienza con una letra, y en tanto los caracteres sucesivos sean letras o dígitos, forman parte del nombre del identificador. Un entero es simplemente una serie de dígitos. La palabra reservada **if** es sólo la letra *i* seguida de la letra *f*. En todos estos casos, existe un modelo sencillo, llamado *autómata de estados finitos* (FSA; *finite state automaton*) o una *máquina de estado* que reconoce esta clase de componentes léxicos. En tanto se sepa en qué estado se está, se puede determinar si la entrada que se ha visto es parte del componente léxico que se está buscando.

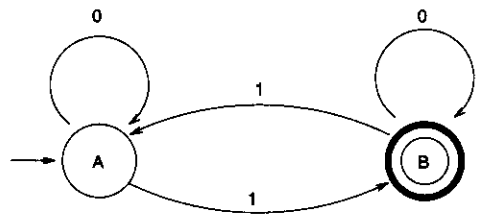


Figura 3.9. FSA para reconocer un número impar de unos.

La figura 3.9 describe un FSA que reconoce un número impar de unos. A partir del estado A (que contiene un arco de entrada que no proviene de otro estado), cada entrada sucesiva "lleva" la máquina al estado A o al estado B haciendo circular el arco marcado con el símbolo de entrada siguiente. Siempre que se está en el estado B (un *estado final* indicado por un círculo doble), la cadena hasta el símbolo vigente es válida y la máquina la acepta. Siempre que se está en el estado A, la cadena no se acepta.

La operación de la máquina para la entrada 100101 procedería como sigue:

Entrada	Estado actual	¿Cadena aceptada?
nulo	A	no
1	B	sí
10	B	sí
100	B	sí
1001	A	no
10010	A	no
100101	B	sí

Se puede ver que, luego de las entradas 1, 10, 100 y 100101, se está en el estado B, se tiene un número impar de unos y se acepta la entrada.

En general, un FSA tiene un estado inicial, un estado final o más, y un conjunto de transiciones (rotuladas como arcos) de un estado a otro. Cualquier cadena que lleve a la máquina del estado inicial a un estado final a través de una serie de transiciones es aceptada por la máquina.

La figura 3.10 es otro ejemplo que define un FSA para que reconozca enteros con signo. En la BNF extendida de la sección anterior, esto sería [+ | –] *dígito* { *dígito* }*. Este ejemplo también demuestra otra característica importante. Existe una dualidad entre máquinas abstractas y gramáticas. Un FSA puede tener como modelo una gramática, como se describirá más adelante en este capítulo.

Autómatas finitos no deterministas. En el análisis que se ha hecho hasta aquí de los autómatas finitos, se ha supuesto implícitamente que todas las transiciones son especificadas de manera exclusiva por el estado actual y el próximo símbolo de entrada. Es decir, para cada estado del

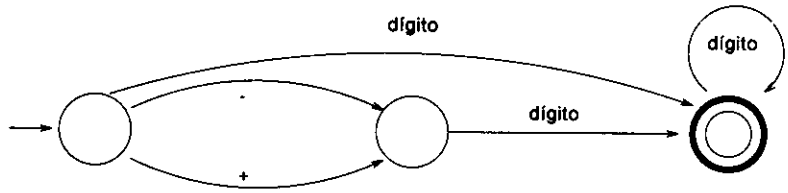


Figura 3.10. FSA para reconocer enteros con signo optativo.

FSA y para cada símbolo de entrada se tiene una transición exclusiva hacia el mismo estado o uno diferente. Un FSA de esta clase se conoce como un FSA *determinista*. Si hay n estados y el alfabeto de entrada tiene k símbolos, entonces el FSA tendrá $n \times k$ transiciones (arcos rotulados). La figura 3.10 constituye un FSA determinista de este tipo. Los arcos faltantes no son el problema —siempre es posible agregar arcos a un estado absorbente de error no final—. El problema del no determinismo es la presencia de múltiples arcos a partir de un estado con el mismo rótulo, de modo que se tiene opción en cuanto al camino a seguir.

Se define un autómata *no determinista* de estados finitos como un FSA con:

1. Un conjunto de estados (nodos en una gráfica)
2. Un estado de partida (uno de los nodos)
3. Un conjunto de estados finales (un subconjunto de los nodos)
4. Un alfabeto de entrada (rótulos en los arcos entre nodos)
5. Un conjunto de arcos de nodos a nodos, cada uno rotulado con un elemento del alfabeto de entrada

A partir de un nodo particular cualquiera puede haber el número que sea (o ninguno) de arcos que van a otros nodos, incluso arcos múltiples con el mismo rótulo. Esta clase de transiciones son no deterministas, puesto que existen varios movimientos posibles para ese símbolo de entrada particular (por ejemplo, véase la figura 3.12). En este caso, se dice que el FSA no determinista *acepta* una cadena si hay algún camino desde el nodo de partida a uno de los nodos finales, aunque pueda haber otros caminos que *no* llegan a un estado final. En el caso determinista, siempre se terminaría en un estado específico que se puede determinar a partir de los símbolos de entrada. Véase el problema 12 para conocer comentarios adicionales sobre los FSA.

Gramáticas normales

Las *gramáticas normales* son casos especiales de gramáticas BNF que resultan equivalentes a los lenguajes de FSA recién descritos. Estas gramáticas tienen reglas de la forma:

$$\langle \textit{no terminal} \rangle ::= \langle \textit{terminal} \rangle \langle \textit{no terminal} \rangle \mid \langle \textit{terminal} \rangle$$

Cada regla consiste en un terminal que puede ir seguido de manera optativa por otro no terminal. Una gramática para generar cadenas binarias que terminan en 0 está dada por:

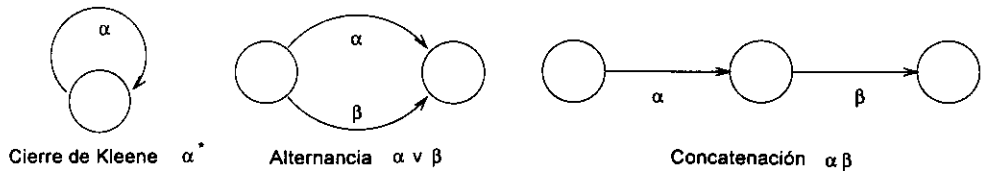


Figura 3.11. Conversión de expresiones normales en un FSA.

$$A \rightarrow 0A \mid 1A \mid 0$$

Las dos primeras alternativas se usan para generar cualquier cadena binaria, y la tercera alternativa se utiliza para concluir la generación con un 0 final.

Existe una asociación estrecha entre FSA y gramáticas normales; se puede demostrar que ambas generan el mismo conjunto de lenguajes. (Véanse los problemas al final del capítulo.)

Expresiones normales

Las *expresiones normales* representan una tercera forma de lenguaje que es equivalente al FSA y la gramática normal. Se define una expresión normal recursivamente como sigue:

1. Los símbolos terminales individuales son expresiones normales.
2. Si *a* y *b* son expresiones normales, entonces también lo son: *a* V *b*, *ab*, (*a*) y *a**.
3. Nada más es una expresión normal.

ab representa la concatenación o puesta en secuencia de las expresiones normales *a* y *b*, *a* V *b* representa la alternancia de *a* o *b*. *a** se conoce como el cierre de Kleene de la expresión normal *a* y representa cero o más repeticiones de *a*, es decir, la cadena nula (que se suele escribir como ∈), *a*, *aa*, *aaa*, ...

Se pueden usar expresiones normales para representar cualquier lenguaje definido por una gramática normal o FSA, aunque la conversión de cualquier FSA a una expresión normal no siempre es obvia. Por ejemplo:

Lenguaje	**Expresión normal**
Identificadores	*letra*(*letra* V *dígito*)*
Cadenas binarias divisibles entre 2	(0 V 1)*0
Cadenas binarias que contienen 01	(0 V 1)*01(0 V 1)*

La conversión de cualquier expresión normal a un FSA es bastante fácil. El operador V representa simplemente caminos alternativos para ir del estado *A* al estado *B*, la concatenación representa una serie de estados, y el cierre de Kleene representa lazos (figura 3.11). El FSA para la tercera expresión normal precedente está dado por la figura 3.12. Adviértase que este proceso da por resultado un FSA no determinista que, como se sabe, también se puede convertir en un FSA determinista equivalente (Problema 12).

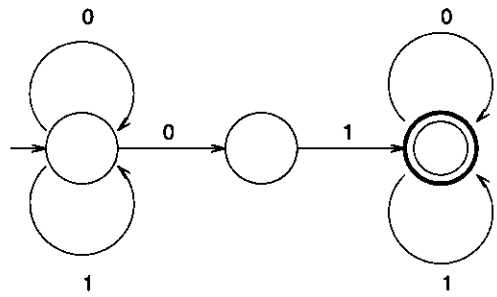


Figura 3.12. Conversión de (0 V 1)*01(0 V 1) en un FSA.

Poder de cómputo de un FSA

Los autómatas de estados finitos tienen una cantidad finita de información, el conjunto de estados. Por consiguiente, el conjunto de cadenas que pueden reconocer es limitado. Por ejemplo, el conjunto de $a^n b^n$ no puede ser reconocido por ningún FSA.

Para ver esto con claridad, supóngase que $a^n b^n$ es reconocido por un FSA con k estados. Para cualquier entrada $a^n b^n$ donde $n > k$, la máquina debe pasar al mismo estado al menos dos veces mientras lee el símbolo a. Por tanto, algún subconjunto de esta cadena causa un lazo dentro del FSA. Esto significa que $a^n = wxy$ donde la subcadena x causa un lazo de regreso al mismo estado en el FSA. Es fácil ver que el FSA acepta a wx^*yb^n. Éstas serán cadenas de la forma $a^{m+i\times p}b^n$ para los enteros m y p y para todos los i, lo cual no es lo mismo que $a^n b^n$.

3.3.3 Autómatas de desplazamiento descendente

En la sección anterior se describió el autómata de estados finitos y se afirmó que el conjunto de lenguajes que acepta un FSA de este tipo era equivalente a los lenguajes generados por gramáticas normales. Existe aquí cierta dualidad. De manera similar, se usan gramáticas BNF para generar cadenas en un lenguaje y se puede utilizar una máquina abstracta para reconocer que una cadena dada está en el lenguaje. En este caso, se puede crear una máquina llamada *autómata de desplazamiento descendente* que equivale a las gramáticas BNF antes estudiadas.

Un *autómata de desplazamiento descendente* (PDA, *pushdown automaton*) es una máquina modelo abstracta similar al FSA. También tiene un conjunto finito de estados. Sin embargo, tiene además una pila de desplazamiento descendente. Los movimientos del PDA son como sigue:

1. Se lee un símbolo de entrada y se lee el símbolo superior de la pila.
2. Con base en ambas entradas, la maquina pasa a un nuevo estado y escribe cero o más símbolos sobre la pila de desplazamiento descendente.

3. La aceptación de la cadena ocurre si la pila está vacía en algún momento. (Alternativamente, la aceptación se puede dar si el PDA se halla en un estado final. Se puede demostrar que ambos modelos son equivalentes.)

Es fácil ver que estos PDA son más poderosos que los FSA de la sección anterior. Las cadenas como $a^n b^n$, que un FSA no puede reconocer, son fácilmente reconocibles para el PDA. Basta con apilar simplemente los símbolos a iniciales y, por cada b, sacar un a de la pila. Si se alcanza el final de la entrada al mismo tiempo que la pila queda vacía, la cadena se acepta.

No es tan claro el hecho de que los lenguajes que aceptan los PDA son equivalentes a los lenguajes libres del contexto. Sin embargo, considérese el proceso de producir la derivación de la extrema izquierda de una cadena. En estos casos, la forma sentencial se puede "guardar" sobre la pila. La operación del PDA es como sigue:

1. Si la parte superior de la pila es un símbolo terminal, compárese con la próxima entrada y sáquese de la pila si es igual. Es un error si los símbolos no coinciden.

2. Si la parte superior de la pila es un símbolo no terminal X, sustitúyase X en la pila por alguna cadena α, donde α es el lado derecho de alguna producción $X \rightarrow \alpha$.

Este PDA simula ahora la derivación de extrema izquierda para cierta gramática libre del contexto. Esta construcción desarrolla en efecto un PDA no determinista que es equivalente a la gramática BNF correspondiente. En el paso 2 de la construcción puede haber más de una regla de la forma $X \rightarrow \alpha$, y no queda claro cuál se debe usar. Se define un *autómata no determinista de desplazamiento descendente* de manera similar al FSA no determinista. Una cadena se acepta si existe una serie posible de movimientos que acepta la cadena.

Si se compara con el caso del FSA, ¿cuál es la relación entre PDA deterministas y PDA no deterministas? En este caso son diferentes. Considérese el conjunto de palíndromos, es decir, las cadenas que se leen igual hacia adelante que hacia atrás, que genera la gramática:

$$S ::= 0S0 \mid 1S1 \mid 2$$

Se puede reconocer esta clase de cadenas por medio de un PDA determinista simplemente:

1. Apile todos los 0 y 1 conforme se lean.
2. Pase a un nuevo estado al leer un 2.
3. Compare cada nueva entrada con la parte superior de la pila y remueva de la pila.

Sin embargo, considérese el conjunto de palíndromos que sigue:

$$S ::= 0S0 \mid 1S1 \mid 0 \mid 1$$

En este caso, nunca se sabe dónde está el punto medio de la cadena. Para reconocer estos palíndromos, el autómata debe adivinar dónde está el punto medio de la cadena. Como ejemplo, dado el palíndromo 011010110, el autómata podría:

Pila:	P. medio:	Equiparar pila con:
	0	11010110
0	1	1010110
01	1	010110
011	0	10110
0110	1	0110
01101	0	110
011010	1	10
0110101	1	0
01101011	0	

Sólo la quinta opción, donde la máquina adivina que 0110 es la primera mitad, concluye satisfactoriamente. Si alguna serie de suposiciones conduce a un análisis sintáctico completo de la cadena de entrada, entonces la cadena es válida de acuerdo con la gramática.

Se ha encontrado que los PDA deterministas son equivalentes a las gramáticas LR(k) y tienen importancia en el diseño de compiladores prácticos para lenguajes de programación. Casi todos los lenguajes con base en gramáticas utilizan una gramática LR(k) para su sintaxis. El estudio completo de las gramáticas LR(k) está fuera del alcance de este libro; sin embargo, en la sección siguiente se describe en forma breve el análisis sintáctico descendente como un ejemplo sencillo del análisis sintáctico de lenguajes libres del contexto.

3.3.4 Algoritmos de análisis sintáctico eficiente

Ya se ha descrito en este capítulo, en forma breve, la estructura general de un analizador sintáctico que descompone una serie de componentes léxicos que representan un programa fuente en el árbol de análisis sintáctico correspondiente. Una gramática BNF para el lenguaje es útil porque define con precisión cuáles secuencias de entrada son programas válidos y también describe una estructura apropiada para cada programa válido. Sin embargo, la gramática describe la estructura de un programa en una forma "de arriba abajo", a partir de un programa completo que se descompone en subprogramas y programa principal, cada uno de estos en enunciados y declaraciones, y así sucesivamente. El analizador sintáctico debe comenzar con los caracteres individuales y construir la estructura desde abajo, trabajando de izquierda a derecha conforme se alimentan los caracteres. Peor aún, los caracteres pueden no formar incluso un programa válido.

Estrategias generales de análisis sintáctico. Desde la época del trabajo de Chomsky se ha comprendido que cada tipo de gramática formal está relacionada estrechamente con un tipo de autómata, una máquina abstracta sencilla que se define por lo regular como capaz de leer una cinta de entrada que contiene una serie de caracteres y producir una cinta de salida que contiene otra serie de caracteres. Por desgracia, surge un problema: puesto que una gramática BNF puede ser ambigua, el autómata debe ser *no determinista*, es decir, puede tener varias opciones de movimientos por hacer y debe "adivinar" cuál es la más apropiada en cualquier momento dado.

Los autómatas no deterministas de desplazamiento descendente pueden reconocer cualquier gramática libre del contexto utilizando esta estrategia de "conjetura". Sin embargo, para la traducción de lenguajes de programación, se necesita un autómata más restringido que nunca tiene que adivinar y que se conoce como autómata *determinista*.

Aunque para las gramáticas normales siempre hay un autómata determinista correspondiente, para las gramáticas BNF no lo hay a menos que la gramática no sea ambigua y satisfaga también otras restricciones. Para gramáticas BNF no ambiguas se han descubierto técnicas sencillas de análisis sintáctico. Una de las primeras técnicas prácticas fue el descenso recursivo, el cual se describe en forma breve más adelante. Un avance importante se dio con el descubrimiento, por parte de Knuth, de una clase de gramáticas llamadas *gramáticas LR* [o algoritmos de análisis sintáctico de izquierda (*left*) a derecha (*right*)], las cuales describen todas las gramáticas BNF reconocidas por autómatas deterministas de desplazamiento descendente. Las gramáticas LR(1) describen todas las gramáticas donde sólo es necesario mirar un símbolo adelante para tomar una decisión de análisis sintáctico. La SLR (Simple LR) y LALR (Lookahead LR) son subclases de gramáticas LR que conducen a algoritmos de análisis sintáctico eficiente. Además, una técnica alternativa de arriba abajo llamada LL, como generalización del descenso recursivo, también proporciona una alternativa práctica de análisis sintáctico. Casi todos los lenguajes actuales se proyectan para ser ya sea SLR, LR o LL, y se pueden usar herramientas generadoras de analizadores sintácticos, como YACC, para generar automáticamente el analizador sintáctico si se da la gramática.

Análisis sintáctico por descenso recursivo

Queda fuera del alcance de este libro la cobertura del espectro completo de algoritmos de análisis sintáctico; sin embargo, el descenso recursivo es relativamente sencillo de describir e implementar, y un ejemplo de analizador sintáctico de descenso recursivo sirve para mostrar la relación entre la descripción formal de un lenguaje de programación y la capacidad para generar código ejecutable para programas escritos en el lenguaje.

Recuérdese que se suele poder reescribir una gramática usando BNF extendida. Por ejemplo, para la sintaxis de enunciados de asignación de la figura 3.7, se afirmó que una expresión aritmética se describía como:

$$\langle \textit{expresión aritmética} \rangle ::= \langle \textit{término} \rangle \ \{[+ \mid -] \ \langle \textit{término} \rangle\}^*$$

Esto expresa que primero se reconoce un ⟨*término*⟩, y luego, en tanto el símbolo siguiente sea + o –, se reconoce otro ⟨*término*⟩. Si se supone que la variable *nextchar* siempre contiene el primer carácter del no terminal respectivo y que la función *getchar* lee en un carácter, entonces se puede reescribir directamente la regla precedente de BNF extendida como el procedimiento recursivo siguiente:

Identificador y *Número* son funciones por leer en las categorías específicas que usen un revisor de estados finitos.

```
procedure AsignarEnun;
   begin
   Variable
   if nextchar <> '=' then Error
      else begin
      nextchar := getchar;
      Expresión
      end
   end;
procedure Expresión;
   begin
   Término;
   while ((nextchar='+') o (nextchar='–')) do
      begin
      nextchar := getchar;
      Término
      end
   end;
procedure Término;
   begin
   Primario
   while ((nextchar='x') o (nextchar='/')) do
      begin
      nextchar := getchar;
      Primario
      end
   end;
procedure Primario;
   begin if nextchar = letra then Variable
   else if nextchar = dígito then Número
   else if nextchar = '(' then
      begin
      nextchar := getchar;
      Expresión;
      if nextchar = ')' then
         nextchar :=getchar
         else Error /* faltante')' */
      end
   else Error /* faltante'(' */
   end;
procedure Variable;
   begin Identificador;
   if nextchar = '[' then
      begin
      nextchar :=getchar
      SubLista;
      if nextchar = ']' then
         nextchar :=getchar
      else Error /* faltante']' */
      end
   end;
procedure SubLista
   begin Expresión;
   while nextchar = ',' do
      begin
      nextchar :=getchar;
      Expresión
      end
   end;
```

Figura 3.13. Analizador sintáctico de descenso recursivo para enunciados aritméticos.

```
procedure Expression;
begin
   Term; /* Llamar término de procedimiento para hallar primer término */
   while ((nextchar='+') o (nextchar='–')) do
   begin
      nextchar := getchar; /* Pasar por alto operador */
      Term
   end
end
```

La figura 3.13 muestra un analizador sintáctico de descenso recursivo completo para la gramática de enunciados de asignación de la figura 3.7. Para terminar el problema de análisis sintáctico, según se expone más adelante en la sección 6.2.1, sólo es necesario darse cuenta de que, dada la expresión *término*$_1$ + *término*$_2$, el sufijo para la misma es simplemente *término*$_1$

término$_2$ +. Como se mostrará más adelante, la conversión del enunciado fuente de entrada en sufijo permite una estrategia de ejecución fácil de implementar para la expresión. Dados los procedimientos para reconocer expresiones aritméticas, el sufijo se puede producir casi con la misma facilidad. Supóngase que cada procedimiento produce el sufijo para su propia subexpresión usando la *salida* del procedimiento. El sufijo para el procedimiento de expresión se puede producir como sigue:

```
procedure Expression;
begin var PlusType: char;
   Term; /* Llamar término de procedimiento para hallar primer término */
   while ((nextchar='+') o (nextchar='-')) do
   begin
      PlusType := nextchar; nextchar := getchar;
      Term; output(PlusType)
   end
end
```

Cada uno de los otros procedimientos se puede modificar de manera similar. El sufijo para *variable* = *expresión* es simplemente *expresión variable* =, el sufijo para *factor*$_1$ × *factor*$_2$ es simplemente *factor*$_1$ *factor*$_2$ ×, y así sucesivamente.

3.3.5 Modelado semántico

Un manual para un lenguaje de programación debe definir el significado de cada construcción en el lenguaje, tanto sola como en conjunto con otras construcciones de lenguaje. El problema es bastante parecido al problema de definición de sintaxis. Un lenguaje suministra una variedad de construcciones diferentes, y tanto el usuario como el implementador del lenguaje requieren una definición precisa de cada construcción. El programador necesita la definición para poder escribir programas correctos y para ser capaz de predecir el efecto de la ejecución sobre cualquier enunciado del programa. El implementador necesita la definición para poder construir una implementación correcta del lenguaje.

En casi todos los manuales de lenguajes, la definición de semántica se da en prosa ordinaria. Típicamente, se da una regla (o conjunto de reglas) de una BNF u otra gramática formal para definir la sintaxis de una construcción, y luego se proporcionan unos cuantos párrafos y algunos ejemplos para definir la semántica. Por desgracia, la prosa suele tener un significado ambiguo, de manera que los diferentes lectores se llevan distintas interpretaciones de la semántica de una construcción de lenguaje. Un programador puede entender mal lo que un programa va a hacer cuando se ejecute, y un implementador puede implementar una construcción de manera diferente que otros implementadores del mismo lenguaje. Como en el caso de la sintaxis, se necesita algún método para proporcionar una definición legible, precisa y concisa de la semántica de un lenguaje completo.

El problema de la definición semántica ha sido objeto de estudio teórico durante tanto tiempo como el problema de la definición sintáctica, pero ha sido mucho más difícil encontrar

una solución satisfactoria. Se han desarrollado muchos métodos diferentes para la definición formal de la semántica. Los siguientes son algunos de ellos:

Modelos gramaticales. En algunos de los primeros intentos para agregar semántica a un lenguaje de programación intervino la adición de extensiones a la gramática BNF que definía el lenguaje. Dado un árbol de análisis sintáctico para un programa, se podía extraer información adicional de ese árbol. En breve se analizarán las gramáticas de atributos como una forma de extraer esta información adicional.

Modelos imperativos u operativos. Una definición *operativa* de un lenguaje de programación es una definición que especifica cómo se ejecutan los programas en el lenguaje en una computadora virtual. Típicamente, la definición de la computadora virtual corresponde a la de un autómata, pero uno que es mucho más complejo que los modelos de autómatas simples que se usaron en el estudio de la sintaxis y el análisis sintáctico. El autómata tiene un estado interno que corresponde al estado interno de un programa cuando se está ejecutando; es decir, el estado contiene todos los valores de las variables, el programa ejecutable mismo y las diversas estructuras de datos de mantenimiento definidas por el sistema. Se usa un conjunto de operaciones formalmente definidas para especificar cómo puede cambiar el estado interno del autómata, en correspondencia con la ejecución de una instrucción del programa. Una segunda parte de la definición especifica cómo se traduce un texto de programa a un estado inicial para el autómata. A partir de este estado inicial, las reglas que definen el autómata especifican cómo pasa el autómata de un estado a otro hasta que se alcanza un estado final. Una definición operativa de un lenguaje de programación como ésta puede representar una abstracción bastante directa de cómo se podría implementar efectivamente el lenguaje, o puede representar un modelo más abstracto que podría constituir la base para un intérprete de software para el lenguaje, pero no para una implementación de producción real.

El *Lenguaje de la Definición de Viena* (VDL; *Vienna Definition Language*) es un enfoque operativo de los años setenta. Amplía el árbol de análisis sintáctico para incluir también el intérprete de máquina. Un *estado* de un cómputo es el árbol de programa y también un árbol que describe todos los datos de la máquina. Cada enunciado lleva el estado del árbol a otro estado de árbol.

Modelos aplicativos. Una definición aplicativa de un lenguaje intenta construir de manera directa una definición de la función que computa cada programa escrito en el lenguaje. Esta definición se construye jerárquicamente a través de la definición de la función que computa cada construcción de programa individual. Correspondiente a los lenguajes aplicativos que se describieron en el capítulo 2, este método es un enfoque aplicativo hacia el modelado semántico.

Cada operación primitiva y definida por el programador que hay en un programa representa una función matemática. Las estructuras de control de secuencia del lenguaje se pueden usar para integrar estas funciones en secuencias más grandes, representadas en el texto del programa por expresiones y enunciados. Las series de enunciados y la bifurcación condicional se representan fácilmente como funciones construidas a partir de la funciones que representan sus componentes individuales. La función que se representa por una iteración se define por lo común de manera recursiva, como una función recursiva construida a partir de los componentes

del cuerpo de la iteración. En último término, se obtiene un modelo funcional completo de todo el programa. El método de *semántica denotativa* de Scott y Strachey y el método de *semántica funcional* de Mills son ejemplos de este enfoque hacia la definición semántica. En la sección 9.4.1 se presenta una breve introducción a la semántica denotativa.

Modelos axiomáticos. Este método amplía el cálculo de predicados para incluir programas. Se puede definir la semántica de cada construcción sintáctica en el lenguaje como axiomas o reglas de inferencia que se pueden usar para deducir el efecto de la ejecución de esa construcción. Para entender el significado del programa completo, se usan los axiomas y reglas de inferencia un poco como en las pruebas ordinarias en matemáticas. A partir del supuesto inicial de que los valores de las variables de entrada satisfacen ciertas restricciones, los axiomas y reglas de inferencia se pueden usar para deducir las restricciones que satisfacen los valores de otras variables después de la ejecución de cada enunciado de programa. En último término, se prueba que los resultados del programa satisfacen las restricciones deseadas de sus valores en relación con los valores de entrada. Es decir, se prueba que los valores de salida representan la función correcta computada a partir de los valores de entrada. El método de *semántica axiomática* desarrollado por Hoare es un ejemplo de este método (sección 9.4.2).

Modelos de especificación. En el *modelo de especificación* se describe la relación entre las diversas funciones que implementan un programa. En tanto se pueda demostrar que una implementación obedece esta relación entre cualesquiera dos funciones, se afirma que la implementación es correcta respecto a la especificación.

El *tipo algebraico de datos* es una forma de especificación formal. Por ejemplo, al construir un programa que implementa pilas, *empilar* y *desempilar* son inversos en el sentido de que, dada cualquier pila P, si se empila algo en P y luego se desempila P, se obtiene de nuevo la pila original. Esto se podría expresar como el axioma:

$$desempilar(empilar(S, x)) = S$$

De cualquier implementación que conserve esta propiedad (junto con varias más) se puede decir que es una implementación correcta de una pila. En la sección 9.4.3 se presenta una breve introducción a los tipos algebraicos de datos.

Las definiciones de semántica formal se están convirtiendo en una parte aceptada de la definición de un nuevo lenguaje. La definición estándar de PL/I incluye una notación de tipo VDL para describir el comportamiento semántico de los enunciados en PL/I, y se desarrolló una definición de semántica denotativa para Ada. Sin embargo, el efecto de los estudios sobre definiciones de semántica formal en la práctica de la definición de lenguajes no ha sido tan fuerte como el efecto del estudio de las gramáticas formales sobre la definición de la sintaxis. Ningún método individual de definición de semántica ha resultado útil tanto para el usuario como para el implementador de un lenguaje. Los métodos operativos pueden proporcionar un buen modelo formal de implementación que el implementador puede usar, pero estas definiciones son por lo común demasiado detalladas para tener algún valor para el usuario. Los métodos funcionales y denotativos no ofrecen mucha orientación al implementador y han probado ser por lo común demasiado complejos para ser directamente valiosos para los usuarios. Los modelos axiomáticos son más comprensibles de manera inmediata para el usuario de lenguajes, pero no se pueden usar en forma general para definir

un lenguaje por completo sin que se vuelvan en extremo complejos, además de que no ofrecen orientación al implementador.

La gramática de atributos se describe en resumen como una forma de modelo semántico de un lenguaje de programación. En la sección 9.4 se describen otros modelos semánticos de lenguajes.

Gramáticas de atributos

Uno de los primeros intentos para desarrollar un modelo semántico de un lenguaje de programación fue el concepto de *gramáticas de atributos*, desarrollado por Knuth [KNUTH 1968]. La idea era asociar una función con cada nodo del árbol de análisis sintáctico de un programa dando el contenido semántico de ese nodo. Las gramáticas de atributos se crearon agregando funciones (*atributos*) a cada regla de una gramática.

Un atributo *heredado* es una función que relaciona valores no terminales de un árbol con valores no terminales más arriba en el árbol. O, en otras palabras, el valor funcional para los no terminales a la derecha de cualquier regla son una función del no terminal del lado izquierdo.

Un atributo *sintetizado* es una función que relaciona el no terminal del lado izquierdo con los valores de los no terminales del lado derecho. Estos atributos pasan información hacia arriba del árbol, es decir, fueron "sintetizados" a partir de la información de la parte baja del árbol.

Considérese esta gramática sencilla para expresiones aritméticas:

$$\begin{aligned} E &\rightarrow T \mid E+ T \\ T &\rightarrow P \mid T\times P \\ P &\rightarrow I \mid (E) \end{aligned}$$

Se puede definir la "semántica" de este lenguaje por medio de un conjunto de relaciones entre los no terminales de la gramática. Por ejemplo, las funciones siguientes producen el valor de cualquier expresión generada por esta gramática:

Producción	**Atributo**
$E \rightarrow E + T$	$valor(E_1) = valor(E_2) + valor(T)$
$E \rightarrow T$	$valor(E) = valor(T)$
$T \rightarrow T \times P$	$valor(T_1) = valor(T_2) \times valor(P)$
$T \rightarrow P$	$valor(T) = valor(P)$
$P \rightarrow I$	$valor(P) = valor\ del\ número\ I$
$P \rightarrow (E)$	$valor(P) = valor(E)$

donde se rotularon X_1 y X_2 para ser la primera o segunda referencia del no terminal X de la producción. La figura 3.14 muestra un árbol con atributos que da el valor de la expresión $2 + 4 \times (1 + 2)$.

Las gramáticas de atributos se pueden usar para transmitir información semántica por todo el árbol sintáctico. Por ejemplo, las producciones de declaraciones de un lenguaje pueden recoger información de declaraciones y esa información de tabla de símbolos se puede transmitir hacia abajo del árbol para usarse en la generación de código para expresiones.

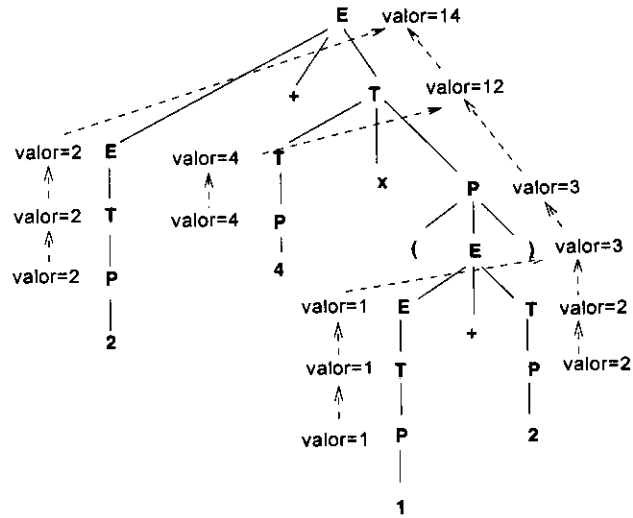


Figura 3.14. Ejemplo de valores de atributos.

Por ejemplo, los atributos siguientes se pueden agregar a los no terminales ⟨*decl*⟩ y ⟨*declaración*⟩ para crear un conjunto de nombres declarados en un programa:

Producción	**Atributo**
⟨*declaración*⟩ ::= ⟨*decl*⟩⟨*declaración*⟩	*decl_conjunto*(*declaración*$_1$) = *decl_nombre*(*decl*) ∪*decl_conjunto* (declaración$_2$)
⟨*declaración*⟩ ::= ⟨*decl*⟩	*decl_conjunto*(*declaración*) = *decl_nombre*(*decl*)
⟨*decl*⟩ ::= **declarar** *x*	*decl_nombre*(*decl*) = *x*
⟨*decl*⟩ ::= **declarar** *y*	*decl_nombre*(*decl*) = *y*
⟨*decl*⟩ ::= **declarar** *z*	*decl_nombre*(*decl*) = *z*

El atributo sintetizado *decl_conjunto* asociado con el no terminal ⟨*declaración*⟩ contendrá siempre el conjunto de nombres que se declaran en cualquier programa particular. Este atributo se puede transmitir luego hacia abajo del árbol por intermedio de un atributo heredado y usarse para ayudar a generar el código correcto para esos datos (véase el Problema 13).

Si una gramática tiene sólo atributos sintetizados (como en este ejemplo), entonces el compilador los puede evaluar al mismo tiempo que se genera el árbol sintáctico durante la fase de análisis sintáctico de la traducción. Así es como funcionan los sistemas como YACC. Cada vez que YACC determina una regla de producción de BNF por aplicar, se ejecuta una subrutina (es decir, su función de atributos) para aplicar la semántica al árbol sintáctico.

3.4 LECTURAS ADICIONALES SUGERIDAS

La literatura sobre la sintaxis y traducción de lenguajes de programación es extensa. Los libros de Aho, Sethi y Ullman [AHO *et al.* 1988] y Fischer y LeBlanc [FISCHER y LEBLANC

1988] proporcionan detalles del proceso de traducción. El capítulo 10 de [BIERMANN 1990] ofrece una perspectiva sencilla del proceso de traducción. El diseño de compiladores de diagnóstico se explica en la descripción de los lenguajes CORC de la Universidad Cornell [CONWAY y MAXWELL 1963] y la variante PL/I del PL/C [CONWAY y WILCOX 1973]. Morgan [MORGAN 1970] analiza la corrección ortográfica.

Las consideraciones prácticas en el diseño de la sintaxis de lenguajes también han recibido atención. Sammet [SAMMET 1969] examina gran parte del material pertinente. Los documentos de fundamentos, como el de Ada [ISO 1994], permiten comprender las decisiones que se tomaron al desarrollar estos lenguajes.

3.5 PROBLEMAS

1. Considere las reglas de gramática BNF siguientes:

$$\begin{aligned} \langle pop \rangle &::= [\langle bop \rangle, \langle pop \rangle] \mid \langle bop \rangle \\ \langle bop \rangle &::= \langle boop \rangle \mid (\langle bop \rangle) \\ \langle boop \rangle &::= x \mid y \mid z \end{aligned}$$

   Para cada una de las cadenas de la lista siguiente, indique todas las categorías sintácticas de la cuales es miembro, en su caso:

   (a) z
   (b) (x)
   (c) [y]
   (d) ([x,y])
   (e) [(x),y]
   (f) [(x),[y,x]]

2. Proporcione una gramática no ambigua que genere el mismo lenguaje que:

$$S \rightarrow SS \mid (S) \mid ( )$$

3. Demuestre que un árbol de análisis sintáctico dado puede ser resultado de múltiples derivaciones (por ejemplo, demuestre que hay varias derivaciones que generan el árbol que muestra la figura 3.4). ¿Qué puede usted decir acerca del árbol de análisis sintáctico si la gramática no es ambigua? ¿Si la gramática es ambigua?

4. Defina una *derivación de extrema izquierda* como una derivación donde se aplica una producción al no terminal de la extrema izquierda de cada forma sentencial. Demuestre que una gramática es ambigua si y solo si alguna cadena del lenguaje tiene dos derivaciones de extrema izquierda definidas.

5. Escriba una gramática BNF para el lenguaje compuesto por todos los números binarios que contienen al menos tres 1 consecutivos. (El lenguaje incluye las cadenas 011101011, 000011110100 y 1111110, pero no 0101011.)

6. La sintaxis del lenguaje *mono* es bastante simple, aunque sólo los monos lo pueden hablar sin cometer errores. El alfabeto del lenguaje es {*a*, *b*, *d*, #} donde # representa un espacio. La gramática es:

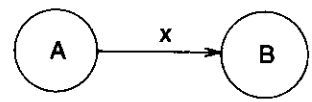
⟨*alto*⟩ ::=	*b*|*d*
⟨*oclusiva*⟩ ::=	⟨*alto*⟩*a*
⟨*sílaba*⟩ ::=	⟨*oclusiva*⟩ | ⟨*oclusiva*⟩⟨*alto*⟩ |*a* ⟨*oclusiva*⟩ |*a* ⟨*alto*⟩
⟨*palabra*⟩ ::=	⟨*sílaba*⟩ | ⟨*sílaba*⟩ ⟨*palabra*⟩ ⟨*sílaba*⟩
⟨*oración*⟩ ::=	⟨*palabra*⟩ | ⟨*oración*⟩#⟨*palabra*⟩

De los oradores siguientes, ¿cuál es el agente secreto que se hace pasar por un mono?

Simio:	*ba # ababadada # bad # dabbada*
Chimpancé:	*abdabaadab # ada*
Babuino:	*dad # ad # abaadad # badadbaad*

7. Proporcione expresiones normales para:

(a) Cadenas binarias que terminan en 01
(b) Enteros decimales divisibles entre 5
(c) Identificadores de C
(d) Cadenas binarias compuestas de ya sea un número impar de unos o un número impar de ceros



Transición de FSA

A ⟶ xB

Regla de gramática normal

Figura 3.15. Conexión entre FSA y gramática

8. Demuestre que cualquier FSA se puede representar por medio de una gramática normal, y que cualquier gramática normal puede ser reconocida por un FSA. La clave está en asociar cada no terminal de la gramática con un estado del FSA. Por ejemplo, la transformación de la figura 3.15 se convierte en la regla $A \rightarrow xB$. (¿Cómo manejaría usted los estados finales?)

9. Proporcione el autómata de estados finitos y la gramática normal para lo siguiente:

(a) (*ab* V *ba*)* V (*ab*)*
(b) (11*)*(110 V 01)
(c) Todas las cadenas arriba de {0, 1} que contengan la cadena 010
(d) Todas las cadenas arriba de {0, 1} que **no** contengan la cadena 010

10. Escriba gramáticas BNF extendidas para los lenguajes definidos por los FSA de la figuras 3.9 y 3.10.

11. Construya árboles de análisis sintáctico para los enunciados de asignación siguientes usando la gramática BNF de la figura 3.3:

(a) A[2] := B + 1
(b) A[I,J] := A[J,I]
(c) X := U – V × W + X / Y
(d) P := U / (V /(W / X))

12. La definición de FSA no determinista incluye el FSA determinista como caso especial. Sin embargo, suponga que tiene un FSA no determinista *N*. Demuestre que existe un FSA determinista *D* que acepta el mismo lenguaje. Esto demuestra que el no determinismo en el caso de los FSA no aumenta el poder de cómputo del modelo. Para hacer esto, en *N* considere el conjunto de estados que se pueden alcanzar a partir de un estado dado y un símbolo de entrada dado. Llame a este conjunto de estados $Estado_i$. Habrá 2^n de estos subconjuntos para *N* con *n* estados. Esto forma el conjunto de estados para el FSA determinista.

13. Agregue a la gramática de la figura 3.3 las siguientes reglas BNF.

⟨*programa*⟩ ::=	⟨*declaración*⟩ ⟨*lista enun*⟩
⟨*declaración*⟩ ::=	⟨*decl*⟩ ⟨*declaración*⟩ | ⟨*decl*⟩
⟨*decl*⟩ ::=	**declarar** ⟨*variable*⟩ ;
⟨*lista enun*⟩ ::=	⟨*enunciado de asignación*⟩ ⟨*lista enun*⟩ | ⟨*enunciado de asignación*⟩

Desarrolle una gramática de atributos tal que:

(a) (*enunciado de asignación*)*válido* = *cierto* si y solo si toda variable del enunciado ha sido declarada.

(b) (*programa*)*válido* = *cierto* si y solo si todo enunciado de asignación del programa es válido.

14. Demuestre que el lenguaje que genera la gramática siguiente:

$$S \rightarrow aSa|a$$

es un lenguaje normal.

15. Un palíndromo es una cadena que se lee igual hacia adelante que hacia atrás.

(a) Demuestre que un conjunto de palíndromos de longitud impar con base en el alfabeto {a, b} es un lenguaje libre del contexto.

(b) Demuestre que el conjunto de cadenas de longitud impar, con base en el alfabeto {a, b}, que **no** son palíndromos está libre del contexto.

(c) ¿Por qué el conjunto de palíndromos de longitud impar requiere un autómata no determinista de desplazamiento descendente para ser reconocido, en vez de un autómata determinista de desplazamiento descendente?

16. Sea S un conjunto normal (es decir, reconocido por un autómata de estados finitos). Demuestre que S^I (S invertido, es decir, el conjunto de cadenas de S escritas hacia atrás) es un conjunto normal.

17. Sea N un conjunto normal. Sea $N^{1/2}$ las "primeras mitades" de N. Es decir, si w está en N y w es la longitud $2k$, entonces los primeros k caracteres de w están en $N^{1/2}$. Demuestre que $N^{1/2}$ es un conjunto normal. (Sugerencia: Despeje S^I primero en el problema anterior.)

representación particular para las secciones. Cualquier cambio en la representación invalidará esas otras partes. Suele ser difícil determinar cuáles subprogramas dependen de una representación particular para objetos de datos si no existe encapsulamiento, y es por ello que los cambios en una representación de datos dan origen a errores sutiles en otras partes del programa que en apariencia no deberían verse afectadas por el cambio.

Los subprogramas constituyen un mecanismo básico de encapsulamiento que está presente en casi todos los lenguajes. Los mecanismos que permiten el encapsulamiento de definiciones de tipos de datos completas son más recientes y, de los lenguajes que aquí se presentan, aparecen sólo en Ada y C++. Adviértase que el ocultamiento de información es primordialmente una cuestión de *diseño de programas*; este ocultamiento es posible en cualquier programa proyectado correctamente, no importa qué lenguaje de programación se use. Sin embargo, el encapsulamiento es en primer término una cuestión de *diseño de lenguajes*; una abstracción se encapsula de manera eficaz sólo cuando el lenguaje prohíbe el acceso a la información oculta en la abstracción.

5.2 ENCAPSULAMIENTO POR SUBPROGRAMAS

Un subprograma es una operación abstracta definida por el programador. Los subprogramas conforman el bloque básico de construcción a partir del cual se construyen la mayoría de los programas, y se encuentran facilidades para su definición e invocación en casi todos los lenguajes. Aquí son importantes dos puntos de vista respecto a los subprogramas. En el nivel de *diseño de programas*, se puede preguntar acerca del sentido en el cual un subprograma representa una operación abstracta que el programador define, en contraposición a las operaciones primitivas que están integradas en el lenguaje. En el nivel de *diseño de lenguajes*, la preocupación se refiere al diseño e implementación de las facilidades generales para definición e invocación de subprogramas. Aunque estos puntos de vista se traslapan, resulta útil tratarlos por separado.

5.2.1 Los subprogramas como operaciones abstractas

Como en el caso de las operaciones primitivas, una definición de subprograma tiene dos partes, una *especificación* y una *implementación*. Sin embargo, para un subprograma ambas partes son suministradas por el programador cuando se define el subprograma.

Especificación de un subprograma. Puesto que un subprograma representa una operación abstracta, deberíamos ser capaces de entender su especificación sin comprender cómo se implementa. La especificación para un subprograma es la misma que para una operación primitiva. Incluye:

1. El *nombre* del subprograma.
2. La *signatura* (también llamada *prototipo*) del subprograma, que da el número de *argumentos*, su orden y el tipo de datos de cada uno, así como el número de *resultados*, su orden y el tipo de datos de cada uno.

3. La *acción* que lleva a cabo el subprograma, es decir, una descripción de la función que calcula.

Un subprograma representa una función matemática que hace corresponder cada conjunto de argumentos con un conjunto particular de resultados. Si un subprograma devuelve explícitamente un objeto de datos de un solo resultado, por lo común se le conoce como una *función de subprograma* (o simplemente *función*), y una sintaxis representativa para su especificación es, en C:

```
float FN(float X, int Y)
```

que especifica la signatura:

$$FN : real \times entero \rightarrow real$$

Adviértase que la especificación también incluye los nombres X y Y, por medio de los cuales se puede hacer referencia a los argumentos dentro del subprograma; estos *parámetros formales* y el tema general de transmisión de parámetros a un subprograma se tratan en la sección 7.3.1. Además, ciertos lenguajes también incluyen una palabra clave en la declaración, como **procedure** o **function**, como en este caso en Pascal:

```
function FN(X: real; Y: integer): real;
```

Si un subprograma devuelve más de un resultado, o si modifica sus argumentos en vez de devolver resultados de manera explícita, se le conoce ordinariamente como un *procedimiento* o *subrutina*, y una sintaxis representativa para su especificación es, en C:

```
void Sub(float X, int Y, float *Z, int *W);
```

En esta especificación, *void* indica una función nula, un subprograma que no devuelve un valor. Un nombre de parámetro formal precedido por * puede indicar un valor de resultado o un argumento que se puede modificar. (Éstos son, de hecho, argumentos apuntador, como se analizará más cabalmente en la sección 7.3.1.) La sintaxis de Ada para esta especificación aclara estas distinciones:

```
procedure Sub(X: in REAL; Y: in integer;
              Z: in out REAL; W: out BOOLEAN)
```

Este encabezado especifica un subprograma con la signatura:

$$Sub : real_1 \times entero \times real_2 \rightarrow real_3 \times booleano$$

Las marcas **in**, **out** e **in out** distinguen las tres maneras de invocar argumentos a un subprograma: **in** designa un argumento que no es modificado por el subprograma, **in out** designa un argumento que se puede modificar y **out** designa un resultado. Estas ideas se tratan de manera más extensa en la sección 7.3.1.

Si bien un subprograma representa una función matemática, surgen problemas cuando se intenta describir con precisión la función computada:

1. Un subprograma puede tener *argumentos implícitos* en forma de variables no locales a las cuales hace referencia.
2. Un subprograma puede tener *resultados implícitos* (*efectos colaterales*) devueltos como cambios a variables no locales o como cambios en sus argumentos de entrada-salida.
3. Un subprograma puede no estar definido para ciertos argumentos posibles, de modo que no completa la ejecución en la forma ordinaria si se le dan esos argumentos; en vez de ello, transfiere el control a algún manejador externo de excepciones (capítulo 6) o termina abruptamente la ejecución del programa completo.
4. Un subprograma puede ser *sensible al historial*, de modo que sus resultados dependen de los argumentos dados a lo largo de todo el historial pasado y no sólo de los argumentos dados en una sola llamada. La sensibilidad al historial se puede deber a que el subprograma retiene datos locales entre invocaciones.

Implementación de un subprograma. Un subprograma representa una operación de la capa de computadora virtual construida por el programador; por tanto, un subprograma se implementa usando las estructuras de datos y operaciones que suministra el mismo lenguaje de programación. La implementación está definida por el *cuerpo* del subprograma, el cual se compone de *declaraciones de datos locales* que definen las estructuras de datos que usa el subprograma y los *enunciados* que definen las acciones que se deben adoptar cuando se ejecuta el subprograma. Ordinariamente, las declaraciones y enunciados se encapsulan para que ni los datos locales ni los enunciados estén disponibles por separado para el usuario del subprograma; el usuario sólo puede invocar el subprograma con un conjunto particular de argumentos y recibir los resultados del cómputo. La sintaxis de C para el cuerpo de un subprograma es representativa:

```
float FN(float X, int Y)      - Signatura de subprograma
   {float M(10); int N;       - Declaraciones de objetos de datos locales
   .                          - Serie de enunciados que definen
   . }                              las acciones del subprograma
```

En ciertos lenguajes (por ejemplo, Pascal, Ada, pero no en C) el cuerpo también puede incluir definiciones de otros subprogramas que representan operaciones definidas por el programador que se usan sólo dentro del subprograma mayor. Estos subprogramas locales también se encapsulan para que no puedan ser invocados desde afuera del subprograma mayor.

Cada invocación de un subprograma requiere argumentos de los tipos correctos, según se establece en la especificación del subprograma. También se deben conocer los tipos de los resultados que devuelve un subprograma. Las cuestiones de verificación de tipos son similares a las correspondientes para operaciones primitivas. La verificación de tipos se puede efectuar en forma estática, durante la traducción, si están dadas las declaraciones para los tipos de los argumentos y resultados de cada subprograma. Alternativamente, la verificación de tipos puede ser dinámica, durante la ejecución del programa. La implementación del lenguaje también puede proporcionar *coerción* de argumentos para convertirlos a los tipos apropiados de manera automática. Estos problemas y métodos de implementación son generalizaciones sencillas de los conceptos presentados en el capítulo 4 para operaciones primitivas. La diferencia princi-

pal proviene de la necesidad de que el programador declare explícitamente información acerca de tipos de argumentos y resultados que es implícita para operaciones primitivas. Sin embargo, una vez que se suministra esta información, los problemas de verificación de tipos se tratan en forma similar.

5.2.2 Definición e invocación de subprogramas

El diseño de recursos para definición e invocación de subprogramas es un problema fundamental, quizá *el* problema fundamental, en el diseño de casi todos los lenguajes. Gran parte de la estructura global de implementación está determinada por la estructura del subprograma en el lenguaje. Aquí se tratan algunos conceptos generales. En el capítulo 8 se examinarán métodos para proporcionar pleno encapsulamiento y ocultamiento de información de datos.

Definiciones y activaciones de subprograma

Un programador escribe una *definición* de subprograma como una propiedad estática de un programa. Durante la ejecución del programa, si se llama (o *invoca*) el subprograma, se crea una *activación* del subprograma. Cuando se completa la ejecución del subprograma, la activación se destruye. Si se hace otra llamada, se crea otra activación. A partir de una sola definición de subprograma se pueden crear muchas activaciones durante la ejecución del programa. La definición sirve como una *plantilla* para crear activaciones durante la ejecución.

La distinción entre definiciones y activaciones de subprograma es importante. Una definición es lo que está presente en el programa como está escrito y es la única información disponible durante la traducción (por ejemplo, se conoce el tipo de variables de subprograma, pero no su valor o ubicación (valor r o valor l)). Las activaciones de subprograma existen sólo durante la ejecución del programa. Durante ésta se puede ejecutar código para tener acceso al valor l o valor r de una variable, pero el tipo de una variable puede no estar disponible a menos que el traductor haya guardado esta información en un descriptor de la variable.

Esta distinción es bastante similar a la que existe entre una definición de tipo y un objeto de datos de ese tipo, como se expone en la próxima sección. La definición de tipo se usa como una plantilla para determinar el tamaño y estructura de los objetos de datos de ese tipo. Sin embargo, los objetos de datos mismos se crean comúnmente durante la ejecución, ya sea cuando se entra a un subprograma o al ejecutar una operación de creación como *malloc*. El uso de *malloc* para crear nuevos objetos de datos según se requieren durante la ejecución del programa corresponde al uso de *llamar* para crear nuevas activaciones de subprograma conforme se les necesita. De hecho, una activación de subprograma *es* un tipo de objeto de datos. Se representa como un bloque de almacenamiento que contiene ciertos elementos de datos componentes que son importantes para la activación del programa. Debe asignarse almacenamiento cuando se crea, y el almacenamiento queda libre cuando se destruye. Una activación tiene un *tiempo de vida*, el tiempo durante la ejecución entre la *llamada* que la crea y el *retorno* que la destruye. Sin embargo, hay conceptos en torno a las activaciones de subprograma para las cuales no existe una analogía directa con otros objetos de datos, por ejemplo, el concepto de *ejecutar* una activación y el concepto de referencia y modificación de otros objetos de datos durante esta ejecución. A causa de estas diferencias, y porque

intuitivamente se hace una marcada distinción en casi todos los lenguajes entre subprogramas y otros objetos de datos, el término *objeto de datos* no se usa aquí para referirse a una activación de subprograma.

Implementación de definición e invocación de subprogramas

Considérese esta definición de subprograma como podría aparecer en C:

```
float FN( float X, int Y)
    {const valinic=2;
    #define valfinal 10
    float M(10); int N;
    ...
    N = valinic;
    if(N<valfinal){ ... }
    return (20 * X + M(N)); }
```

Esta definición especifica los componentes necesarios para una activación del subprograma en tiempo de ejecución:

1. El renglón de signatura de *FN* suministra información para almacenamiento de *parámetros* (los objetos de datos *X* y *Y*) y almacenamiento para el *resultado de la función*, un objeto de datos de tipo *float*.
2. Las declaraciones tienen providencias de almacenamiento para *variables locales* (arreglo *M* y variable *N*).
3. Almacenamiento para *literales* y *constantes definidas*. *Valinic* es la constante definida 2, *valfinal* es la constante definida 10, y 10 y 20 son literales.
4. Almacenamiento para el *código ejecutable* generado a partir de los enunciados del cuerpo del subprograma.

Adviértase una característica importante de C. El atributo **const** informa al compilador de C que el objeto de datos *valinic* tiene el valor de literal del entero 2. Sin embargo, el enunciado **#define** es un mandato de preprocesador (macro) que convierte cada aparición de *valfinal* en los caracteres "10". El traductor de C nunca procesa el nombre *valfinal*. Los efectos prácticos de cada enunciado sobre el programa ejecutable son los mismos, pero sus significados son bastante diferentes. *Valinic* tendrá un valor l cuyo valor r es 2, en tanto que *valfinal* tendrá sólo un valor r de 10.

La definición del subprograma permite organizar estas áreas de almacenamiento y determinar el código ejecutable durante la traducción. El resultado de la traducción es la plantilla que se usa para construir cada activación particular en el momento en que el subprograma es llamado durante la ejecución. La figura 5.1 muestra la definición de subprograma traducida a una plantilla en tiempo de ejecución.

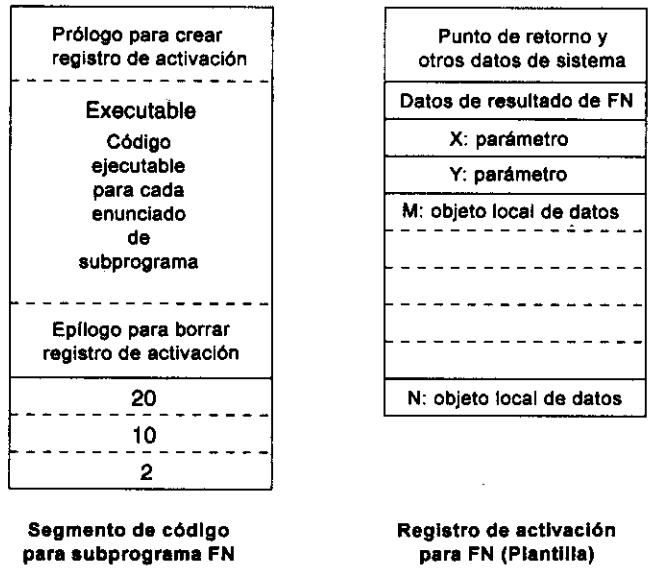


Figura 5.1. Estructura de una activación de subprograma.

Para construir una activación particular del subprograma a partir de su plantilla, la plantilla completa se podría copiar en una nueva área de memoria. Sin embargo, en vez de hacer una copia completa, es mucho mejor dividir la plantilla en dos partes:

- Una parte estática, llamada *segmento de código*, compuesta de las constantes y el código ejecutable anteriores. Esta parte no debe variar durante la ejecución del subprograma, y de esta manera una sola copia puede ser compartida por todas las activaciones.

- Una parte dinámica, llamada *registro de activación*, compuesta de los parámetros, resultados de función y datos locales anteriores, más otros elementos de datos de "mantenimiento" definidos por la implementación, como áreas de almacenamiento temporal, puntos de retorno y vinculaciones para referencia de variables no locales (que se analizan en la sección 7.2). Esta parte tiene la misma estructura para cada activación, pero contiene diferentes valores de datos. En esta forma, cada activación tiene necesariamente su propia copia de la parte del registro de activación.

En la figura 5.2 se muestra la estructura resultante durante la ejecución. Para cada subprograma, existe un solo segmento de código almacenado a lo largo de la ejecución del programa. Los registros de activación se crean y destruyen dinámicamente durante la ejecución cada vez que se llama el subprograma y cada vez que el mismo concluye con un retorno.

El tamaño y estructura del registro de activación que se requiere para un subprograma se puede determinar ordinariamente durante la traducción; por ejemplo, el compilador (o traductor) puede determinar cuántos componentes se necesitan para guardar los datos necesarios

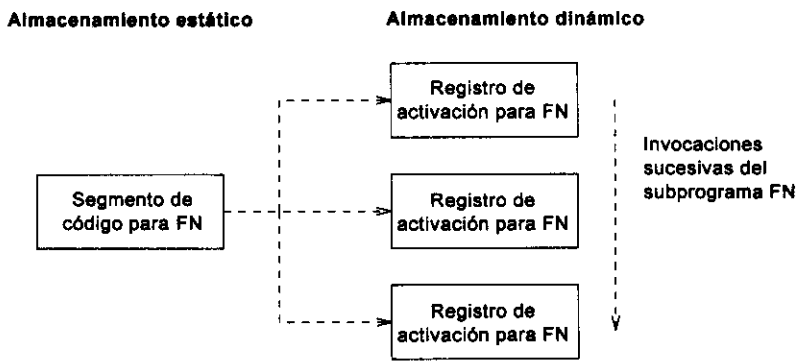


Figura 5.2. Código compartido y registros de activación por separado.

en el registro de activación y la posición de cada componente en el mismo. Se puede entonces tener acceso a los componentes usando el cálculo de dirección base más desplazamiento, como se describió en la sección 4.3.3 para objetos de datos de registro ordinarios. Por esta razón, es usual que un registro de activación se represente de hecho en el almacenamiento simplemente como cualquier otro objeto de datos de registro. Para crear un nuevo registro de activación sólo se requiere que se conozca el tamaño del bloque de almacenamiento del registro, en vez de su estructura interna en detalle (puesto que los desplazamientos dentro del bloque ya se han computado durante la traducción y sólo se necesita la dirección base para completar el cálculo de acceso durante la ejecución). En lugar de guardar una plantilla completa de registros de activación en tiempo de ejecución, sólo se debe guardar el tamaño del registro de activación para que lo use la operación de *llamar* al crear el registro de activación. La gestión de almacenamiento en la llamada y retorno de subprogramas implica sólo la asignación de un bloque de tamaño apropiado cuando se llama un subprograma, y la liberación del bloque con el retorno.

Cuando se llama un subprograma, tiene lugar un cierto número de acciones ocultas que se ocupan de establecer el registro de activación, de la transmisión de parámetros, la creación de vínculos para referencia no local y actividades similares de "mantenimiento". Estas acciones deben tener lugar antes de que se ejecute el código real para el enunciado en el cuerpo del subprograma. Por lo común, la ejecución de este *prólogo* antes de la ejecución del bloque de código para el subprograma se maneja a través de la inserción, por parte del traductor, de un bloque de código para ejecutar estas acciones al inicio del subprograma. Cuando un subprograma termina, se requiere un conjunto similar de acciones de mantenimiento para devolver resultados y liberar el almacenamiento destinado al registro de activación. Un *epílogo* es un conjunto de instrucciones que el traductor inserta al final del bloque de código ejecutable para llevar a cabo estas acciones. Más adelante en este capítulo se podrán encontrar más detalles.

Subprogramas genéricos

La especificación de un subprograma menciona ordinariamente el número, orden y tipos de datos de los argumentos. Un *subprograma genérico* es aquel que tiene un solo nombre pero varias definiciones diferentes que se distinguen por una signatura diferente. Se dice, entonces, que el nombre de un subprograma genérico es un *homónimo*. Los conceptos generales de una operación genérica y un nombre de operación homónimo se tratan en el capítulo 4 en relación con las operaciones primitivas. Los subprogramas también pueden ser genéricos en la misma forma. Por ejemplo, al escribir un conjunto de subprogramas para un programa universitario de inscripción a clases pueden ser necesarias dos rutinas, una para introducir una "sección" en una tabla de secciones de clase y otra para introducir un "estudiante" en una lista de clase para una "sección". Ambos subprogramas se podrían definir usando el nombre *Intro*:

```
procedure  Intro(Estudiante: in entero;
                 Secc: in out Sección) is
      begin
           – enunciados para introducir estudiantes en una lista de matrícula
             de sección
      end;
procedure  Intro(S: in Sección;
                 Tab: in out ListaClase) es
      begin
             – enunciados para introducir sección en una lista de clase
      end;
```

El nombre *Intro* es un homónimo y se ha convertido en el nombre de un subprograma genérico *Intro*. Cuando se hace una llamada a *Intro*: *Intro*(*A*,*B*), el traductor debe resolver los tipos de los argumentos *A* y *B* con los tipos apropiados para los dos procedimientos de *Intro*. En este caso, si *A* es de tipo *entero*, la llamada es a la primera definición del subprograma *Intro*, y si *A* es de tipo *sección*, entonces es una llamada a la segunda. Puesto que esta resolución se hace durante la traducción, en un lenguaje como Ada no hay efecto alguno sobre la organización del lenguaje en tiempo de ejecución. Una vez que se ha resuelto la homonimia, la traducción de la llamada al subprograma procede como para cualquier otra llamada a subprograma.

En FORTRAN 90, los procedimientos homónimos se especifican a través de un bloque de INTERFACE. El ejemplo anterior se podría especificar como se muestra en la figura 5.3. En este ejemplo, la subrutina *INTRO* se definió de manera que tome ya sea un parámetro de tipo *entero* o un parámetro de tipo *sección*. Pero esto se puede ampliar para incluir el tipo mismo como un parámetro, igual que en el mecanismo de tipos *polimórficos* del ML. Esto se analizará con mayor detalle en el capítulo 8.

Los subprogramas introducen un cambio no significativo en el lenguaje o la posible implementación de éste, pero tienen gran impacto en cómo el lenguaje logra ser usado. Por lo tanto, lejos de ser una sobrecarga, una característica extra en lenguajes como Ada se convierte en una propiedad central de programación en ML. Esto se estudiará con mayor detalle en el capítulo 8.

```
INTERFACE INTRO
   SUBROUTINE INTRO-ESTUDIANTE(ESTUDIANTE, SECC)
      ENTERO :: ESTUDIANTE
      SECCIÓN :: SECC
      END SUBROUTINE INTRO-ESTUDIANTE
SUBROUTINE INTRO.SECCIÓN(S, TAB)
      SECCIÓN :: S
      LISTACLASE :: TAB
      END SUBROUTINE INTRO-SECCIÓN
END INTERFACE INTRO
```

Figura 5.3. Bloque de interfaz en FORTRAN 90.

5.2.3 Definiciones de subprograma como objetos de datos

En casi todos los lenguajes compilados, como C, FORTRAN y Pascal, la definición del subprograma es independiente de la ejecución del mismo. Primero, un compilador procesa el programa fuente a la forma en tiempo de ejecución. Durante la ejecución, la parte estática de la definición del subprograma es a la vez inaccesible e invisible. Sin embargo, en lenguajes como LISP y Prolog (que son implementados ordinariamente por intérpretes de software), típicamente no existe distinción alguna entre estas dos fases. Los recursos permiten tratar las *definiciones* de subprograma como objetos de datos en tiempo de ejecución en sí mismas.

La *traducción* es una operación que toma una definición de subprograma en forma de una cadena de caracteres y produce el objeto de datos en tiempo de ejecución que representa la definición. La *ejecución* es una operación que toma una definición en la forma de tiempo de ejecución, crea una activación a partir de ella y ejecuta la activación. La operación de ejecución se invoca por medio de la primitiva de *llamada* usual, pero la operación de traducción se suele considerar como una *metaoperación* independiente que tiene lugar para todos los subprogramas antes de que comience la ejecución del programa global. En LISP y Prolog, sin embargo, la traducción también es una operación que se puede invocar en *tiempo de ejecución* sobre un objeto de datos de cadena de caracteres para producir la forma ejecutable de un cuerpo de subprograma. Estos lenguajes suministran una operación *definir* que toma un cuerpo y especificación de subprograma y produce una definición invocable y completa de subprograma (por ejemplo, *define* en LISP, *consult* en Prolog).

Por consiguiente, en ambos lenguajes es posible comenzar la ejecución del programa sin tener un subprograma particular en existencia. Durante la ejecución, el cuerpo del subprograma se puede introducir por lectura o crearlo como un objeto de datos de cadena de caracteres y luego traducirlo a una forma ejecutable. La operación *definir* se puede usar entonces para proporcionar un nombre y definir parámetros para el cuerpo, con lo que se obtiene una definición completa. Posteriormente, el programa se puede llamar según se requiera. Más tarde se puede modificar la definición del subprograma. En esta forma, las definiciones de subprograma se convierten en auténticos objetos de datos en estos lenguajes.

5.4 GESTIÓN DE ALMACENAMIENTO

La gestión de almacenamiento para datos es una de las preocupaciones fundamentales del programador, del implementador de lenguajes y del diseñador de lenguajes. En esta sección se consideran los diversos problemas y técnicas en la gestión de almacenamiento.

Típicamente, los lenguajes contienen muchas características o restricciones que sólo se pueden explicar por un deseo de parte de los encargados del diseño de permitir el uso de una

u otra técnica de gestión de almacenamiento. Tómese, por ejemplo, la restricción en FORTRAN a las llamadas no recursivas de subprograma. Las llamadas recursivas podrían permitirse en FORTRAN sin cambiar la sintaxis, pero su implementación requeriría una pila en tiempo de ejecución de puntos de retorno, una estructura que requiere gestión de almacenamiento dinámico durante la ejecución. Sin llamadas recursivas, FORTRAN se puede implementar con sólo gestión de almacenamiento estático. Constituye una ruptura drástica con el pasado el hecho de que, en el estándar más reciente de FORTRAN 90, se permite almacenamiento dinámico limitado (véase el Capítulo 10). Pascal está proyectado cuidadosamente para permitir gestión de almacenamiento con base en pilas, LISP para permitir recolección de basura, etcétera.

En tanto que todos los diseños de lenguajes permiten ordinariamente el uso de ciertas técnicas de gestión de almacenamiento, los detalles de los mecanismos, y su representación en hardware y software, son tarea del implementador. Por ejemplo, mientras que el diseño de LISP puede implicar una lista de espacios libres y recolección de basura como base apropiada para la gestión de almacenamiento, existen varias técnicas distintas de recolección de basura entre las cuales el implementador deberá elegir con base en el hardware y el software disponibles.

En tanto que el programador también se ocupa a fondo de la gestión de almacenamiento y debe proyectar programas que usen el almacenamiento de manera eficiente, es probable que tenga poco control directo sobre el almacenamiento. Un programa afecta el almacenamiento sólo de manera indirecta a través del uso o falta de uso de diferentes características del lenguaje. Esto se dificulta aún más por la tendencia tanto de los encargados del diseño como de los implementadores de lenguajes a tratar la gestión de almacenamiento como un tema *dependiente de la máquina* que no se debe analizar directamente en los manuales de lenguajes. Por eso, para el programador suele ser difícil descubrir qué técnicas se usan efectivamente.

5.4.1 Elementos principales en tiempo de ejecución que requieren almacenamiento

El programador tiende a ver la gestión de almacenamiento en gran medida en términos de almacenamiento de datos y programas traducidos. Sin embargo, la gestión de almacenamiento en tiempo de ejecución abarca muchas otras áreas. Algunas, como los puntos de retorno para subprogramas, ya se han tocado previamente; otras no se han mencionado todavía en forma explícita. Examinemos los principales elementos de programas y datos que requieren almacenamiento durante la ejecución del programa.

Segmentos de código para programas de usuario traducidos. En cualquier sistema se debe asignar un bloque importante de almacenamiento para guardar los segmentos de código que representan la forma traducida de programas de usuario, sin tomar en cuenta si los programas son interpretados por hardware o por software. En el primer caso, los programas serán bloques de código de máquina ejecutable; en el segundo, estarán en alguna forma intermedia.

Programas de sistema en tiempo de ejecución. Otro bloque considerable de almacenamiento se debe asignar a programas de sistema que apoyan la ejecución de los programas de usuario. Éstos pueden ir desde simples *rutinas de biblioteca*, como funciones de seno, coseno o impresión de cadenas, hasta intérpretes o traductores de software presentes durante la ejecución. También se incluyen aquí las rutinas que controlan la gestión de almacenamiento en

tiempo de ejecución. Estos programas de sistema serían ordinariamente bloques de código de máquina ejecutable por hardware, sin tomar en cuenta la forma ejecutable de los programas de usuario.

Estructuras de datos y constantes definidas por el usuario. Debe asignarse espacio para datos de usuario destinado a todas las estructuras de datos declaradas en programas de usuario o creadas por ellos, incluso constantes.

Puntos de retorno en subprogramas. Los subprogramas tienen la propiedad de que se les puede invocar desde distintos puntos de un programa. Por consiguiente, se debe asignar almacenamiento a la información de control de secuencia generada internamente, como los puntos de retorno de subprogramas, puntos de reanudación de corrutinas o incluso avisos para subprogramas planificados. Como se habrá de señalar en el capítulo 6, el almacenamiento de estos datos puede requerir sólo localidades únicas, una pila central u otra estructura de almacenamiento en tiempo de ejecución.

Entornos de referencia. El almacenamiento de entornos de referencia (asociaciones de identificadores) durante la ejecución puede requerir espacio considerable, como, por ejemplo, la lista A de LISP (capítulo 13).

Temporales en evaluación de expresiones. La evaluación de expresiones requiere el uso de almacenamiento temporal definido por el sistema para los resultados intermedios de evaluación. Por ejemplo, al evaluar la expresión $(x + y) \times (u + v)$, el resultado de la primera suma puede tener que guardarse en un temporal mientras se efectúa la segunda suma. Cuando en las expresiones intervienen llamadas recursivas de función, se puede necesitar un número potencialmente ilimitado de temporales para guardar resultados parciales en cada nivel de recursión.

Temporales en transmisión de parámetros. Cuando se llama un subprograma, se debe evaluar una lista de parámetros reales y guardar los valores resultantes en almacenamiento temporal hasta que se completa la evaluación de toda la lista. Cuando la evaluación de un parámetro puede requerir evaluación de llamadas recursivas de función, se puede necesitar una cantidad potencialmente ilimitada de almacenamiento, como en la evaluación de expresiones.

Buffers de entrada-salida. Ordinariamente, las operaciones de entrada y salida trabajan a través de *buffers* (memorias temporales), que sirven como áreas de almacenamiento temporal donde se guardan datos entre el momento de la transferencia física efectiva de datos a o desde almacenamiento externo y las operaciones de entrada y salida iniciadas por el programa. Suele ser necesario reservar cientos de localidades de memoria para estos *buffers* durante la ejecución.

Datos diversos de sistema. En casi todas las implementaciones de lenguajes, se requiere almacenamiento para diversos datos de sistema: tablas, información de situación para entrada-salida y diversos fragmentos de información de estado (por ejemplo, cuentas de referencia o bits de recolección de basura).

Además de los datos y elementos de programa que requieren almacenamiento, también resulta instructivo examinar las diversas operaciones que pueden requerir asignación o liberación de almacenamiento. Las operaciones principales son:

Operaciones de llamada y retorno de subprogramas. La asignación de almacenamiento para un registro de activación de subprograma, el entorno local de referencia y otros datos al llamar un subprograma suele ser la operación principal que requiere asignación de

*almacen*amiento. La ejecución de una operación de retorno de un subprograma requiere por lo común liberar el espacio asignado durante la llamada.

Operaciones de creación y destrucción de estructuras de datos. Si el lenguaje suministra operaciones que permiten crear nuevas estructuras de datos en puntos arbitrarios durante la ejecución del programa (en vez de sólo al entrar al subprograma), entonces estas operaciones requieren ordinariamente asignación de almacenamiento por separado del asignado al entrar al subprograma. La operación *new* en Pascal y la función *malloc* en C son ejemplos de estas operaciones de creación. El lenguaje también puede suministrar una operación explícita de destrucción, como la *dispose* en Pascal y la función *free* en C, las cuales pueden requerir que se libere almacenamiento.

Operaciones de inserción y eliminación de componentes. Si el lenguaje suministra operaciones que insertan y eliminan componentes de estructuras de datos, se puede requerir asignación y liberación de almacenamiento para implementar estas operaciones, por ejemplo, las operaciones de lista de ML y LISP que insertan un componente en una lista.

Mientras que estas operaciones requieren gestión de almacenamiento explícita, para muchas otras es necesario que tenga lugar cierta gestión de almacenamiento oculta. Gran parte de esta actividad de gestión de almacenamiento implica la asignación y liberación de almacenamiento temporal para propósitos de mantenimiento (por ejemplo, durante la evaluación de expresiones y la transmisión de parámetros).

5.4.2 Gestión de almacenamiento controlada por el programador y por el sistema

¿En qué medida se debe permitir al programador controlar directamente la gestión de almacenamiento? Por una parte, C se ha vuelto muy popular porque permite que el programador ejerza un extenso control sobre el almacenamiento a través de *malloc* y *free*, que asignan y liberan almacenamiento para estructuras de datos definidas por el programador. Por otro lado, muchos lenguajes de alto nivel no permiten al programador control alguno; la gestión de almacenamiento se afecta sólo de manera implícita a través del uso de diversas características del lenguaje.

La dificultad que presenta el control de la gestión de almacenamiento por el programador es doble: puede imponer una carga grande y a menudo indeseable sobre el programador, y también puede interferir con la necesaria gestión de almacenamiento controlada por el sistema. Ningún lenguaje de alto nivel puede permitir que el programador se eche encima la carga completa de la gestión de almacenamiento. Por ejemplo, difícilmente se puede esperar que el programador se ocupe con el almacenamiento de temporales, puntos de retorno de subprograma u otros datos de sistema. En el mejor de los casos, podría controlar la gestión de almacenamiento para datos locales (y quizá programas). No obstante, incluso la simple asignación y liberación de almacenamiento para estructuras de datos, como en C, es probable que dé lugar a la generación de basura y referencias desactivadas. Por tanto, la gestión de almacenamiento controlada por el programador es "peligrosa" para él mismo porque puede conducir a sutiles errores o pérdida de acceso a almacenamiento disponible. La gestión de almacenamiento controlada por el programador también puede interferir con la gestión de almacenamiento controlada por el sistema, en cuanto a que, para el almacenamiento controlado por el pro-

gramador, se pueden requerir áreas de almacenamiento y rutinas de gestión de almacenamiento especiales, lo que ocasiona un uso menos eficiente del almacenamiento en conjunto.

La ventaja de permitir control de gestión de almacenamiento por el programador radica en el hecho de que suele ser extremadamente difícil para el sistema determinar cuándo se puede asignar y liberar almacenamiento con más eficacia. El programador, por otra parte, conoce a menudo con bastante precisión cuándo se necesita una estructura de datos particular o cuándo ya no es necesaria y se puede liberar.

El dilema no es en manera alguna trivial y suele estar en el centro de la decisión sobre qué lenguaje usar en un proyecto dado. ¿Se proporciona "protección" para el programador usando un lenguaje con tipos fuertes y características eficaces de gestión de almacenamiento, con una reducción correspondiente en el rendimiento, o se necesitan las características de rendimiento (por ejemplo, gestión de almacenamiento y velocidad de ejecución) con un aumento en el riesgo de que el programa pueda contener errores y fallar durante la ejecución? Éste es uno de los debates fundamentales en la comunidad de ingeniería de software. Este libro no va a resolver el debate, pero un objetivo importante es proporcionar al lector los detalles apropiados para hacer una elección inteligente en estos asuntos.

Fases de la gestión de almacenamiento

Es conveniente identificar tres aspectos básicos de la gestión de almacenamiento:

Asignación inicial. Al inicio de la ejecución, cada segmento de almacenamiento puede estar ya sea asignado o libre. Si está libre inicialmente, está disponible para asignarse de manera dinámica conforme avanza la ejecución. Todo sistema de gestión de almacenamiento requiere alguna técnica para llevar la cuenta del almacenamiento libre, así como mecanismos para asignación de almacenamiento libre conforme surge la necesidad durante la ejecución.

Recuperación. El almacenamiento que ha sido asignado y usado, y que posteriormente queda disponible, debe ser recuperado por el gestor de almacenamiento para volver a usarlo. La recuperación puede ser muy simple, como en la reubicación de un apuntador de pila, o muy compleja, como en la recolección de basura.

Compactación y nuevo uso. El almacenamiento recuperado puede estar listo de inmediato para volver a usarse, o puede requerirse una compactación para construir bloques grandes de almacenamiento libre a partir de fragmentos pequeños. En el nuevo uso del almacenamiento intervienen ordinariamente los mismos mecanismos que en la asignación inicial.

Se conocen y usan muchas técnicas diferentes de gestión de almacenamiento en las implementaciones de lenguajes. Es imposible examinarlas todas, pero un puñado relativo de ellas basta para representar los enfoques básicos. Casi todas las técnicas son variantes de uno de estos métodos básicos.

5.4.3 Gestión de almacenamiento estático

La forma más sencilla de asignación es la *asignación estática*, que tiene lugar durante la traducción y permanece fija a lo largo de la ejecución. Ordinariamente, el almacenamiento para los segmentos de código de programas de usuario y de sistema se asigna de manera estática, como se hace con el almacenamiento para *buffers* de E/S y diversos datos de sistema. La asignación estática no requiere software de gestión de almacenamiento en tiempo de ejecución y, desde luego, no hay que preocuparse por la recuperación y nuevo uso.

En la implementación usual de FORTRAN, todo el almacenamiento se asigna de manera estática. Cada subprograma se compila por separado y el compilador establece el segmento de código (que incluye un registro de activación) que contiene el programa compilado, sus áreas de datos, temporales, ubicación de puntos de retorno y diversos datos de sistema. El cargador asigna espacio en la memoria para estos bloques compilados en el tiempo de carga, así como espacio para rutinas de sistema en tiempo de ejecución. Durante la ejecución del programa no es necesario que tenga lugar gestión de almacenamiento alguna.

La asignación estática de almacenamiento es eficiente porque no se gasta tiempo ni espacio para gestión de almacenamiento durante la ejecución. El traductor puede generar las direcciones de valor l directas para todos los datos. Sin embargo, es incompatible con llamadas recursivas de subprograma, con estructuras de datos cuyo tamaño depende de datos computados o introducidos, y con muchas otras características de lenguaje deseables. En las subsecciones siguientes de este capítulo se analizarán diversas técnicas para la *gestión de almacenamiento dinámico* (en tiempo de ejecución). Sin embargo, el lector no debe perder de vista la importancia de la asignación estática. Para muchos programas, la asignación estática es bastante satisfactoria. Dos de los lenguajes de programación de uso más extendido, FORTRAN y COBOL, están proyectados para asignación estática de almacenamiento (aunque, como ya se ha señalado, el FORTRAN 90 permite ahora arreglos dinámicos y procedimientos recursivos), y lenguajes, como C, que tienen almacenamiento dinámico, también permiten crear datos estáticos para una ejecución eficiente.

5.4.4 Gestión de almacenamiento con base en pilas

La técnica más simple de gestión de almacenamiento en tiempo de ejecución es la pila. El almacenamiento libre al inicio de la ejecución se establece como un bloque secuencial en la memoria. Conforme se asigna almacenamiento, el mismo se toma de localidades secuenciales en este bloque de pila a partir de un extremo. El almacenamiento se debe liberar en el orden inverso de asignación para que el bloque de almacenamiento que está siendo liberado se encuentre siempre en el tope de la pila.

Un solo *apuntador de pila* es todo lo que se necesita para controlar la gestión de almacenamiento. El apuntador de pila siempre apunta al tope de la pila, la próxima palabra disponible de almacenamiento libre en el bloque de pila. Todo el almacenamiento en uso está en la pila abajo de la ubicación a la que señala el apuntador de pila. Todo el almacenamiento libre se encuentra arriba del apuntador. Cuando se va a asignar un bloque de k localidades, se mueve simplemente el apuntador de modo que apunte k localidades más arriba en el área de la pila. Cuando se libera un bloque de k localidades, se hace retroceder el apuntador k localidades. La *compactación* ocurre en forma automática como parte de la liberación de almacenamiento. Al liberar un bloque de almacenamiento, se recupera en forma automática el almacenamiento liberado y queda disponible para nuevo uso.

Puesto que muchos de los programas y elementos de datos que requieren almacenamiento están ligados a activaciones de subprogramas, en casi todos los lenguajes la estructura estrictamente anidada de último en entrar, primero en salir de las llamadas y devoluciones de subprograma es lo que hace de la gestión de almacenamiento por pila una técnica atractiva. El agrupamiento de estos elementos asociados con una activación de subprograma y que requieren asignación de pila en un solo *registro de activación*, como se expuso en la sección

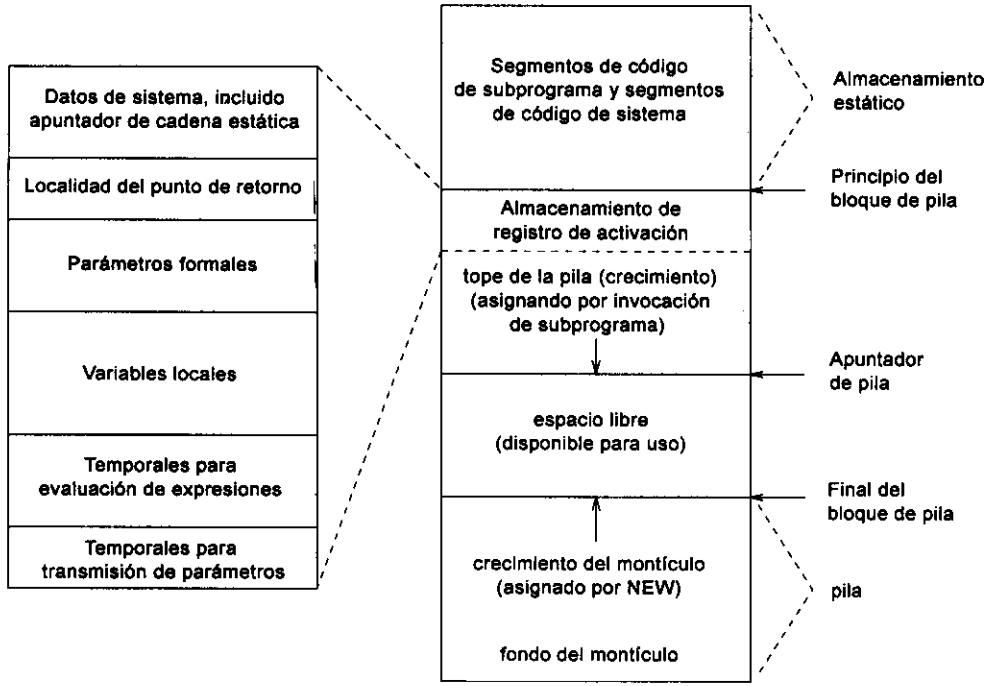


Figura 5.5. Organización de la memoria en Pascal.

5.2, es una técnica común. Cuando se llama un subprograma, se crea un nuevo registro de activación en la cima de la pila; la terminación causa su eliminación de la pila.

Casi todas las implementaciones de Pascal se construyen en torno a una sola pila central de registros de activación para subprogramas junto con un área asignada estáticamente, la cual contiene programas de sistema y segmentos de código de subprogramas. En la figura 5.5(a) se muestra la estructura de un registro de activación típico para un subprograma en Pascal. El registro de activación contiene todos los elementos variables de información asociados con una activación de subprograma dada. La figura 5.5(b) muestra una organización típica de memoria durante la ejecución de Pascal (el área de almacenamiento en montículo se usa para almacenamiento asignado por *new* y liberado por *dispose*).

El uso de una pila en una implementación de LISP es un poco diferente. Las llamadas de subprograma (función) también están estrictamente anidadas y una pila se puede usar para registros de activación. Cada registro de activación contiene un punto de retorno y temporales para evaluación de expresiones y transmisión de parámetros. Los ambientes de referencia también se podrían asignar en la misma pila, excepto que al programador se le permite manipular en forma directa estas asociaciones. Por consiguiente, se acostumbra guardarlos en una pila por separado, representada como una lista vinculada y que se llama *A-list* (*lista A*). La pila que contiene puntos de retorno y temporales se puede ocultar entonces del programador y asignarse de manera secuencial. La implementación de LISP también requiere un área de

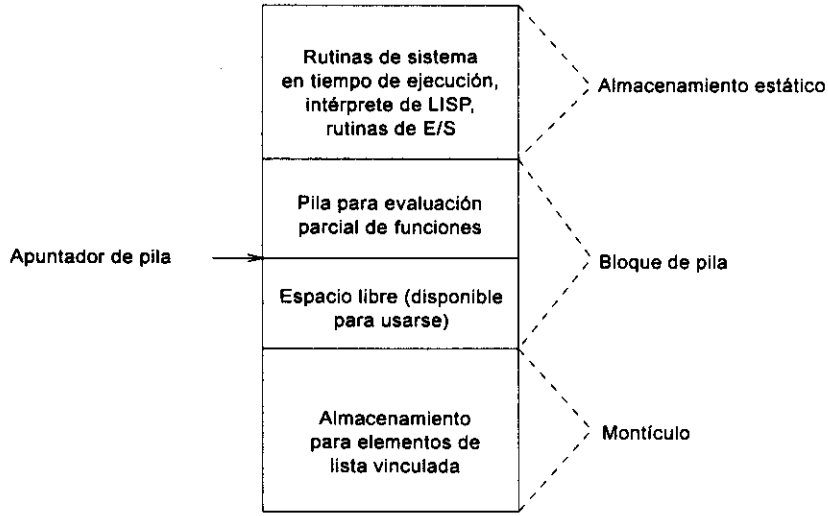


Figura 5.6. Organización de la memoria en LISP durante la ejecución.

almacenamiento en *montículo*, que se administra a través de una lista de espacios libres y recolección de basura, con un área especial y gestor de almacenamiento para elementos de datos de palabra completa, como los números. En la figura 5.6 se muestra una organización típica de la memoria en LISP.

El uso de una pila de registros de activación de subprograma (o registros parciales de activación, como en LISP) es característica de la implementación de todos los lenguajes de la parte II. La gestión de almacenamiento con base en pilas es la técnica de uso más extendido para gestión de almacenamiento en tiempo de ejecución.

La división del almacenamiento en un área asignada estáticamente, un área asignada como una pila y un área asignada como un montículo, que se observa en Pascal y LISP, es característica de un cierto número de lenguajes. Ada, C y Prolog también usan esta división tripartita de la memoria en áreas administradas de diferentes maneras.

5.4.5 Gestión de almacenamiento en montículos: Elementos de tamaño fijo

El tercer tipo básico de gestión de almacenamiento se conoce como *gestión de almacenamiento en montículos*. Un *montículo* es un bloque de almacenamiento dentro del cual se asignan o liberan segmentos de alguna manera relativamente no estructurada. En este caso, los problemas de asignación, recuperación, compactación y nuevo uso de almacenamiento pueden ser graves. No existe una técnica única de gestión de almacenamiento en montículos, sino más bien una colección de técnicas para manejar diversos aspectos de la administración de esta memoria.

La necesidad de almacenamiento en montículos surge cuando un lenguaje permite asignar y liberar almacenamiento en puntos arbitrarios durante la ejecución del programa, como cuando un lenguaje permite la creación, destrucción o extensión de estructuras de datos de programador en puntos arbitrarios. Por ejemplo, en ML se pueden concatenar dos listas para crear una lista

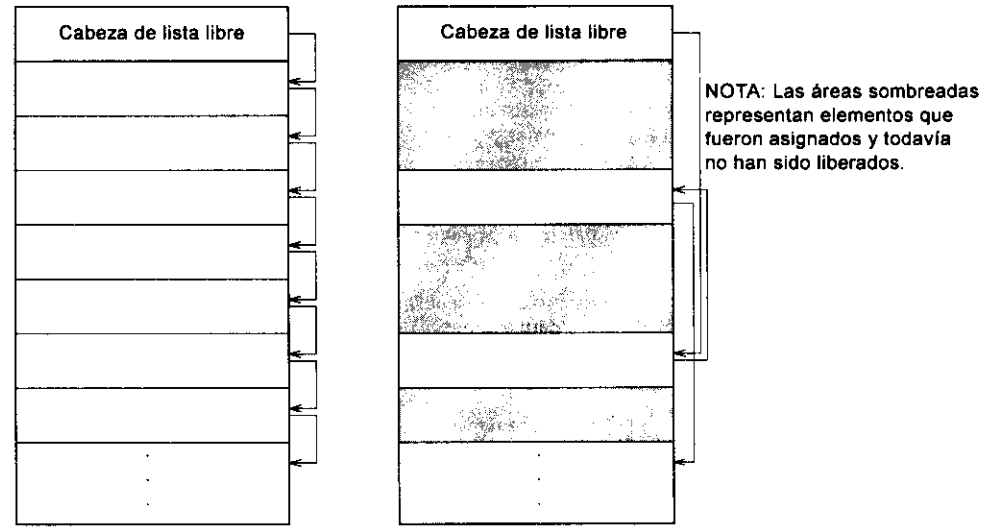


Figura 5.7. Estructura de listas de espacios libres.

nueva en cualquier punto arbitrario durante la ejecución, o el programador puede definir de manera dinámica un tipo nuevo. En LISP, se puede agregar un nuevo elemento a una estructura de lista ya existente en cualquier punto, lo que una vez más requiere asignación de almacenamiento. Tanto en ML como en LISP, también se puede liberar almacenamiento en puntos impredecibles durante la ejecución.

Es conveniente dividir las técnicas de gestión de almacenamiento en montículos en dos categorías, en función de si los elementos asignados son siempre del mismo tamaño fijo o de tamaño variable. Cuando se usan elementos de tamaño fijo, las técnicas de gestión se pueden simplificar en forma considerable. La compactación, en particular, no representa un problema, pues todos los elementos disponibles son del mismo tamaño. En esta sección se examinará el caso con tamaño fijo y se dejará el caso con tamaño variable para la próxima sección.

Supóngase que cada elemento de tamaño fijo que se asigna a partir del montículo y luego se recupera ocupa N palabras de memoria. Típicamente, N puede ser 1 o 2. Si se supone que el montículo ocupa un bloque contiguo de memoria, se divide conceptualmente el bloque de montículo en una serie de elementos K, cada uno de N palabras de largo, de modo que $K \times N$ es el tamaño del montículo. Siempre que se necesita un elemento, se asigna uno de éstos a partir del montículo. Cada vez que se libera un elemento, deberá ser uno de estos elementos originales del montículo.

Inicialmente, los elementos K están vinculados entre sí para formar una *lista de espacios libres* (es decir, la primera palabra de cada elemento de la lista libre apunta a la primera palabra del siguiente elemento de la misma lista). Para asignar un elemento, se elimina de la lista el primer elemento de la lista de espacios libres y se devuelve un apuntador al mismo a la operación que solicita el almacenamiento. Cuando un elemento se libera, se vincula simplemente de nuevo a la cabeza de la lista de espacios libres. La figura 5.7 ilustra una lista de

espacios libres de este tipo, así como la lista después de la asignación y liberación de varios elementos.

Recuperación: Conteos de referencias y recolección de basura

La devolución del almacenamiento recién liberado a la lista de espacios libres es sencilla, siempre y cuando el almacenamiento se pueda identificar y recuperar. Pero la identificación y recuperación pueden ser bastante difíciles. El problema radica en determinar cuáles elementos del montículo están disponibles para nuevo uso y, por tanto, pueden ser devueltos a la lista de espacios libres. Existen tres soluciones de uso bastante extendido.

Devolución explícita por programador o sistema. La técnica de recuperación más simple es la de *devolución explícita*. Cuando un elemento que ha estado en uso queda disponible para usarse de nuevo, se debe identificar explícitamente como "libre" y devolverse a la lista de espacios libres (por ejemplo, una llamada a *dispose* en Pascal). Cuando los elementos se usan para propósitos de sistema, como almacenamiento o ambientes de referencia, puntos de retorno o temporales, o cuando toda la gestión de almacenamiento está controlada por el sistema, cada rutina de sistema es responsable de devolver espacio conforme queda disponible para volver a usarse, a través de la llamada explícita de una rutina de *liberar* con el elemento apropiado como parámetro.

La devolución explícita es una técnica natural de recuperación para almacenamiento en montículos, pero desafortunadamente no siempre es factible. Las razones se hallan en dos viejos problemas: *basura* y *referencias desactivadas*. Estos problemas se analizaron por primera vez en el capítulo 4 en relación con la destrucción de estructuras de datos. Si se destruye una estructura (y se libera el almacenamiento) antes de haber eliminado todas las rutas de acceso a la estructura, toda ruta de acceso remanente se convierte en una referencia desactivada. Por otra parte, si se destruye la última ruta de acceso a una estructura sin deshacer la estructura misma y se recupera el almacenamiento, entonces la estructura se convierte en basura. En el contexto de la gestión de almacenamiento en montículos, una referencia desactivada es un apuntador a un elemento que ha sido devuelto a la lista de espacios libres (que pueden haber sido reasignados para otros fines). Un elemento de basura es aquel que está disponible para nuevo uso pero no se halla en la lista de espacios libres y por ello se ha vuelto inaccesible.

Si la basura se acumula, el almacenamiento disponible se reduce de manera gradual hasta que el programa puede ser incapaz de continuar por falta de espacio libre conocido. Las referencias desactivadas pueden originar un caos. Si un programa intenta modificar a través de una referencia desactivada una estructura que ya ha sido liberada, se puede modificar inadvertidamente el contenido de un elemento de la lista de espacios libres. Si esta modificación sobrescribe el apuntador que vincula el elemento al próximo elemento en la lista de espacios libres, todo el resto de la misma puede volverse defectuoso. Aún peor, un intento posterior por parte del asignador de almacenamiento para usar el apuntador del elemento sobrescrito conduce a resultados completamente impredecibles; por ejemplo, un fragmento de programa ejecutable podría ser asignado como "espacio libre" y modificarse más tarde. Se presentan problemas similares si el elemento al que apunta la referencia desactivada ya ha sido reasignado a otro uso antes que se haga una referencia.

La devolución explícita de un almacenamiento en montículo facilita la creación de basura y referencias desactivadas. En tales casos, es fácil que el programador cree basura o referencias desactivadas de manera involuntaria. Por ejemplo, considérense los enunciados en C:

```
int *p, *q;                    /* p y q son apuntadores a enteros */
...
p = malloc(sizeof(int));       /* asigna un int y fija valor l en p */
p = q;                         /* uf - el valor l antes en p se ha perdido */
```

o

```
int *p, *q;                    /* p y q son apuntadores a enteros */
...
p = malloc(sizeof(int));       /* asigna un int y fija valor l en p */
p = q;                         /* - guarda valor l de espacio asignado en q */
free(p);                       /* uf - referencia desactivada todavía en 9 */
```

Puede ser igualmente difícil para el sistema en tiempo de ejecución evitar la creación de basura o referencias desactivadas. En LISP, por ejemplo, las listas vinculadas son una estructura básica. Una de las operaciones primitivas de LISP es *cdr*, la cual, dado un apuntador a un elemento de una lista vinculada, devuelve un apuntador al elemento siguiente de la lista [véase la figura 5.8(a)]. El elemento al que se apuntaba originalmente *puede* haber sido liberado por la operación *cdr*, siempre y cuando el apuntador original al que se aplicó *cdr* haya sido el único

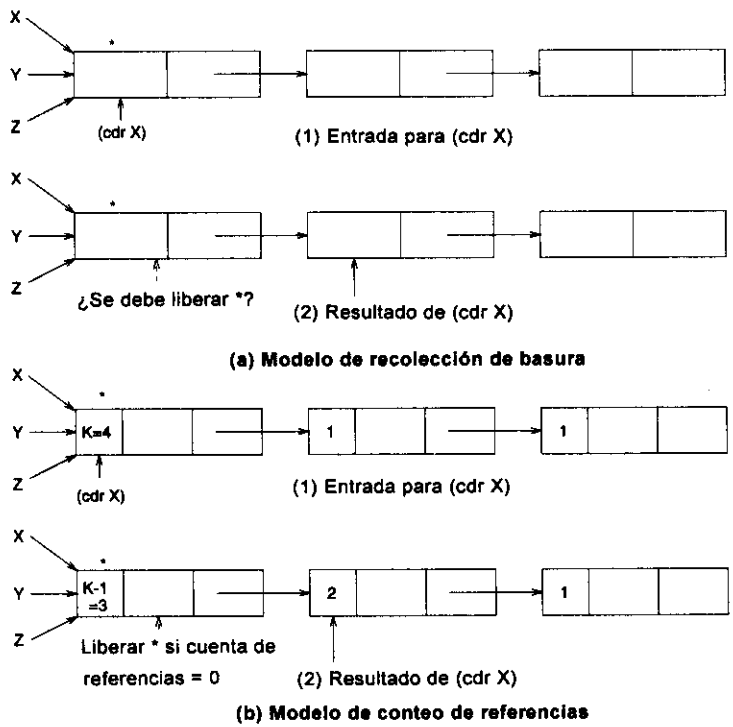


Figura 5.8. Operación *cdr* de LISP.

apuntador al elemento. Si *cdr* no devuelve el elemento a la lista de espacios libres en este punto, se convierte en basura. Sin embargo, si *cdr* devuelve el elemento a la lista y existen otros apuntadores a él, entonces se transforman en referencias desactivadas. Si no existe una forma directa para determinar si existen apuntadores de esta naturaleza, entonces la primitiva *cdr* debe generar potencialmente basura o referencias desactivadas.

Debido a estos problemas con la devolución explícita, son deseables enfoques alternativos. Una alternativa, llamada *conteos de referencias*, requiere devolución explícita pero proporciona una manera de verificar el número de apuntadores a un elemento dado para evitar la creación de referencias colgadas. Una segunda alternativa, llamada *recolección de basura*, consiste en permitir la creación de basura pero no de referencias desactivadas. Más tarde, si la lista de espacios libres se agota, se invoca un mecanismo *recolector de basura* para identificar y recuperar la basura.

Conteos de referencias. El uso de conteos de referencias es la más simple de las dos técnicas. Dentro de cada elemento del montículo se provee cierto espacio adicional para un *contador de referencias*. El contador de referencias contiene el *conteo de referencias* que indica el número existente de apuntadores a ese elemento. Cuando se asigna inicialmente un elemento de la lista de espacios libres, su cuenta de referencias se fija en 1. Cada vez que se crea un nuevo apuntador al elemento, su cuenta de referencias se incrementa en 1. Cada vez que se destruye un apuntador, la cuenta de referencias se reduce en 1. Cuando la cuenta de referencias de un elemento llega a cero, el elemento está libre y se puede devolver a la lista de espacios libres.

Los conteos de referencias permiten evitar tanto basura como referencias desactivadas en casi cualquier situación. Considérese una vez más la operación *cdr* de LISP. Si cada elemento de la lista contiene una cuenta de referencias, entonces para la operación *cdr* es sencillo evitar las dificultades anteriores. La *cdr* debe restar 1 a la cuenta de referencias del elemento al que originalmente apuntaba su entrada. Si el resultado deja una cuenta de referencias de cero, entonces el elemento se puede devolver a la lista de espacios libres y, si no es cero, entonces otros apuntadores todavía apuntan al elemento y no se puede considerar como libre [véase la figura 5.8(b)].

Cuando al programador se le permite un enunciado *free* (*liberar*) o *erase* (*borrar*) explícito, los conteos de referencias también proporcionan protección. Un enunciado *free* disminuye en 1 la cuenta de referencias de la estructura. Sólo si entonces la cuenta es cero, la estructura es devuelta en efecto a la lista de espacios libres. Un conteo de referencias distinto de cero indica que la estructura todavía es accesible y que el mandato *free* se debe pasar por alto.

La dificultad más importante asociada a los conteos de referencias es el costo de mantenerlos. La prueba, incremento y reducción de cuentas de referencias ocurre de manera continua a lo largo de la ejecución, lo cual suele causar una reducción considerable en la eficiencia de ejecución. Considérese, por ejemplo, la asignación simple $P := Q$, donde P y Q son ambas variables apuntador. Sin conteos de referencias, basta con copiar simplemente el apuntador de Q en P. Con cuentas de referencia se tiene que hacer lo siguiente:

1. Obtener acceso al elemento al que apunta P y disminuir su cuenta de referencias en 1.
2. Probar la cuenta resultante; si es cero, devolver el elemento a la lista de espacios libres.
3. Copiar en P el valor l guardado en Q.
4. Obtener acceso al elemento al que apunta Q y aumentar su cuenta de referencias en 1.

El costo total de la operación de asignación ha aumentado en forma considerable. Cualquier operación similar que pueda crear o destruir apuntadores también deberá modificar los conteos de referencias. Además, se tiene el costo del almacenamiento adicional para los conteos de referencias. Cuando para empezar los elementos tienen una longitud de sólo una o dos posiciones, el almacenamiento de conteos de referencias puede reducir sustancialmente el almacenamiento disponible para datos. Sin embargo, esta técnica es popular cuando se tienen sistemas de procesamiento en paralelo, pues el costo de mantener las cuentas de referencias se distribuye entre los usuarios de los datos, mientras que, en el modelo de recolección de basura siguiente, la recuperación de datos se hace bastante costosa.

Recolección de basura. De regreso al problema básico de la basura y las referencias desactivadas, se puede estar fácilmente de acuerdo en que las referencias desactivadas son potencialmente más dañinas que la basura. La acumulación de basura causa una merma en la cantidad de almacenamiento utilizable, pero las referencias desactivadas pueden conducir al caos total debido a la modificación aleatoria del almacenamiento en uso. Por supuesto, ambos problemas están relacionados: las referencias desactivadas se producen cuando el almacenamiento se libera "demasiado pronto", y la basura cuando el almacenamiento no se libera sino "demasiado tarde". Cuando no es factible o resulta demasiado costoso evitar ambos problemas de manera simultánea a través de un mecanismo como los conteos de referencias, la generación de basura es claramente preferible para evitar referencias desactivadas. Es mejor no recuperar el almacenamiento que recuperarlo demasiado pronto.

La filosofía básica que está atrás de la recolección de basura es simplemente permitir la generación de basura para evitar referencias desactivadas. Cuando la lista de espacios libres se agota por completo y se necesita más almacenamiento, el cómputo se suspende temporalmente y se inicia un procedimiento extraordinario, una *recolección de basura*, la cual identifica los elementos en el montículo y los devuelve a la lista de espacios libres. El cómputo original se reanuda entonces, y una vez más se acumula basura hasta que se agota la lista de espacios libres, momento en el cual se inicia otra recolección de basura, y así sucesivamente.

Puesto que la recolección de basura se hace sólo raras veces (cuando la lista de espacios libres se agota), es permisible que el proceso sea bastante costoso. Intervienen dos etapas:

Marcar. En la primera etapa se debe marcar cada elemento del montículo que está activo, es decir, que es parte de una estructura de datos accesible. Cada elemento debe contener un *bit de recolección de basura* que se fija inicialmente en "activado". El algoritmo de marcado fija el bit de recolección de basura de cada elemento activo en "desactivado".

Barrer. Una vez que el algoritmo ha marcado los elementos activos, todos los restantes cuyo bit de recolección de basura está "activado" son basura y se pueden devolver a la lista de espacios libres. Un simple examen secuencial del montículo es suficiente. Se verifica el bit de recolección de basura conforme se le encuentra en el examen. Si está "desactivado", el elemento se pasa por alto; si está "activado", el elemento se vincula dentro de la lista de espacios libres. Todos los bits de recolección de basura se reajustan a "activado" durante el examen (como preparación para una recolección de basura posterior).

La parte de marcado de la recolección de basura es la más difícil. Puesto que la lista de espacios libres está agotada cuando se inicia la recolección de basura, cada elemento en el montículo está activo (es decir, todavía en uso) o es basura. Desafortunadamente, la inspección

EJEMPLO 5.1. Asignación de almacenamiento en LISP.

La relación entre el montículo de LISP y la pila de LISP se muestra en la figura 5.8. En esta figura se supone que:

- El almacenamiento en montículo contiene 15 elementos de los cuales 9 están actualmente en la lista *libre* [figura 5.8(a)]
- Las dos definiciones siguientes han sido introducidas por el usuario:

```
(defun f1(x y z) (cons x (f2 y z)))
(defun f2(v w) (cons v w))
```

La ejecución de la expresión (f1 'a '(b c) '(d e)) tiene lugar como sigue:

1. Se invoca *f*1 y los argumentos *x*, *y* y *z* se agregan a la pila usando las 9 entradas de montículo disponibles de la lista *libre* [figura 5.8(b)].
2. Se invoca *f*2 con apuntadores a sus argumentos *v* y *w* [figura 5.8(c)].
3. La lista *libre* está vacía. El recolector de basura marca primero los elementos a los que se apunta desde la pila y luego, en un segundo paso, pone todos los elementos restantes en la lista *libre* [figura 5.8(d)].
4. Se calcula el valor de *f*2 y se pone en la pila [figura 5.8(e)].
5. Se calcula el valor de *f*1 y se pone en la pila [figura 5.8(f)]. El sistema de LISP mostraría automáticamente este resultado al usuario.
6. Todos los elementos del cómputo son ahora basura. Cuando la lista *libre* vuelva a estar vacía, serán recuperados por la próxima recolección de basura.

de un elemento no puede indicar su situación porque nada hay intrínseco en un elemento basura que indique que ya no es accesible desde otro elemento activo. Más aún, la presencia de un puntero a un elemento desde otro elemento del montículo no indica necesariamente que el elemento al que se apunta está activo; puede ser que ambos elementos sean basura. Así pues, un simple examen del montículo en busca de punteros y que marca los elementos a los que se apunta como activos no es suficiente.

¿Cuándo está activo un elemento del montículo? Es claro que un elemento está activo si existe un apuntador a él desde *afuera del montículo* o desde otro elemento *activo* del montículo. Si es posible identificar todos los punteros externos de esta clase y marcar los elementos apropiados del montículo, entonces se puede iniciar un proceso iterativo de marcado que examina estos elementos activos en busca de apuntadores a otros elementos no marcados. Estos nuevos elementos son marcados entonces y examinados en busca de otros apuntadores, y así sucesivamente. Es necesario un uso bastante disciplinado de los apuntadores porque tres supuestos críticos sustentan este proceso de marcado:

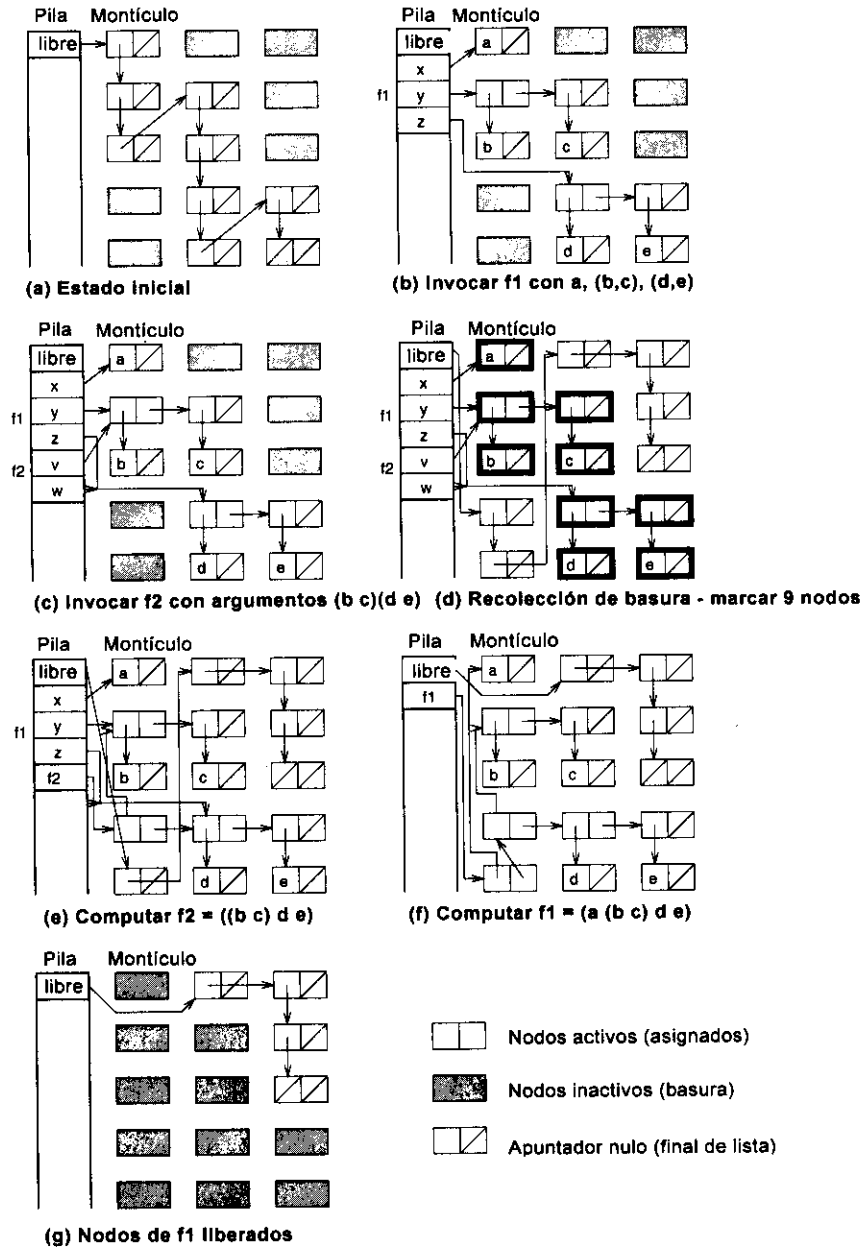


Figura 5.9. Asignación de almacenamiento en montículos y pilas en LISP.

1. Cualquier elemento activo se debe poder alcanzar a través de una cadena de apuntadores que se inicia afuera del montículo.
2. Debe ser posible identificar todos los apuntadores afuera del montículo que señalan a un elemento dentro del montículo.
3. Debe ser posible identificar, dentro de cualquier elemento activo del montículo, los campos que contienen apuntadores a otros elementos del montículo.

El LISP satisface estos tres supuestos, lo cual permite la recolección de basura en datos de LISP. Pero si cualquiera de estos supuestos no se satisface, entonces el proceso de marcado dejará de señalar ciertos elementos activos. Por ejemplo, en C, los supuestos 2 y 3 pueden no ser necesariamente ciertos. Si se usara recolección de basura en C, el resultado podría ser la recuperación de elementos activos y con ello la generación de referencias desactivadas.

Resulta instructiva la manera como estos supuestos se satisfacen en una implementación típica de LISP. Primero, a cada elemento de montículo se le da un formato idéntico, por lo común con dos campos de apuntador y un conjunto de bits adicionales para datos de sistema (incluido un bit de recolección de basura). Como cada elemento de montículo contiene exactamente dos apuntadores, y éstos están siempre en las mismas posiciones dentro del elemento, se satisface el supuesto 3. Segundo, existe sólo un pequeño conjunto de *estructuras de datos de sistema* que pueden contener apuntadores al interior del montículo (la lista A, la lista OB, la lista de desplazamiento descendente, etc.). Se puede garantizar que el marcado iniciado a partir de estas estructuras de datos de sistema permitirá identificar todos los apuntadores externos al interior del montículo, como lo exige el supuesto 2. Por último, es imposible alcanzar un elemento de montículo como no sea a través de una cadena de apuntadores que se inicia afuera del montículo. Por ejemplo, un apuntador a un elemento de montículo no se puede computar por adición de una constante para dar otro apuntador. Por tanto, se satisface el supuesto 1. El ejemplo 5.1 muestra cómo interactúan pila, montículo y recolección de basura en LISP durante la ejecución del programa.

5.4.6 Gestión de almacenamiento en montículos: Elementos de tamaño variable

La gestión de almacenamiento en montículos donde se asignan y recuperan elementos de tamaño variable es más difícil que con elementos de tamaño fijo, aunque se aplican muchos de los mismos conceptos. Se presentan elementos de tamaño variable en múltiples situaciones. Por ejemplo, si se está usando espacio para estructuras de datos definidas por el programador y guardadas en forma secuencial, como arreglos, entonces se van a requerir bloques de espacio de tamaño variable, o se podrían asignar registros de activación para tareas en un montículo, en bloques secuenciales de tamaños variables.

Las dificultades principales con los elementos de tamaño variable tienen que ver con el nuevo uso del espacio recuperado. Incluso si se recuperan dos bloques de espacio de cinco palabras en el montículo, puede ser imposible satisfacer una solicitud posterior de un bloque de seis palabras. Este problema no surgió en el caso más simple de bloques de tamaño fijo; el espacio recuperado siempre se pudo volver a usar de inmediato.

Asignación inicial y nuevo uso

Con elementos de tamaño fijo, era apropiado dividir el montículo de inmediato en un conjunto de elementos y luego basar la asignación inicial en una lista de espacios libres que contenía estos elementos. Esta clase de técnica no es aceptable con elementos de tamaño variable. En vez de ello, se desea mantener el espacio libre en bloques de un tamaño tan grande como sea posible. En un principio, por tanto, se considera al montículo como simplemente un bloque grande de almacenamiento libre. Un *apuntador de montículo* es adecuado para la asignación inicial. Cuando se solicita un bloque de N palabras, el apuntador de montículo se hace avanzar N y el valor original del apuntador de montículo es devuelto como apuntador al elemento recién asignado. Conforme se libera almacenamiento detrás del apuntador de montículo que avanza, se puede recoger en una lista de espacios libres.

En último término, el apuntador de montículo alcanza el final del bloque de montículo. Ahora se debe volver a usar parte del espacio libre que queda atrás en el montículo. Se presentan dos posibilidades de nuevo uso a causa del tamaño variable de los elementos:

1. Usar la lista de espacios libres para asignación, examinando la lista en busca del bloque de tamaño apropiado y devolviendo cualquier espacio sobrante a la lista después de la asignación.
2. Compactar el espacio libre moviendo todos los elementos activos a un extremo del montículo, dejando el espacio libre como un solo bloque en el extremo y restableciendo el apuntador de montículo al principio de este bloque.

Examinemos estas dos posibilidades por turno.

Nuevo uso directamente a partir de una lista de espacios libres

El enfoque más simple, cuando se recibe una solicitud de un elemento de N palabras, consiste en examinar la lista de espacios libres en busca de un bloque de N palabras o más. Un bloque de N palabras se puede asignar directamente. Un bloque de más de N palabras se debe dividir en dos bloques, uno de N palabras, que se asignar de inmediato, y el bloque residual, el cual se devuelve a la lista de espacios libres. Se emplean varias técnicas particulares para administrar directamente la asignación a partir de una lista de espacios libres de este tipo:

1. *Método de primer ajuste.* Cuando se necesita un bloque de N palabras, se examina la lista de espacios libres en busca del *primer* bloque de N palabras o más, el cual se divide luego en un bloque de N palabras y el resto, el cual se devuelve a la lista de espacios libres.
2. *Método de mejor ajuste.* Cuando se necesita un bloque de N palabras, se examina la lista de espacios libres en busca del bloque con el número *mínimo* de palabras mayor que o igual a N. Este bloque se asigna como una unidad, si tiene exactamente N palabras, o se divide y el residuo se devuelve a la lista de espacios libres.

Mantener los elementos de la lista de espacios libres en orden de tamaño, hace que la asignación sea bastante eficiente. Sólo es necesario examinar la lista hasta encontrar el tamaño apropiado que se necesita. Sin embargo, se genera el correspondiente costo adicional de agregar

entradas a la lista de espacios libres por tener que examinar la misma en busca del lugar apropiado para agregar la nueva entrada.

Recuperación con bloques de tamaño variable

Antes de considerar el problema de la compactación de memoria, demos un vistazo a las técnicas para recuperación cuando intervienen bloques de tamaño variable. Es relativamente poco lo que es diferente aquí con respecto a los bloques de tamaño fijo. La devolución explícita de espacio liberado a la lista de espacios libres es la técnica más sencilla, pero los problemas de basura y referencias desactivadas están presentes una vez más. Se puede usar conteo de referencias en la forma ordinaria.

La recolección de basura es también una técnica factible. Sin embargo, surgen ciertos problemas adicionales con bloques de tamaño variable. La recolección de basura tiene lugar como antes, con una fase de marcado seguida de una fase de recolección. El marcado se debe basar en las mismas técnicas de seguimiento de cadena de apuntadores. La dificultad radica ahora en la recolección. Antes se recolectaba a través de un simple examen secuencial de la memoria, probando el bit de recolección de basura de cada elemento. Si el bit estaba "activado", el elemento era devuelto a la lista de espacios libres; si estaba "desactivado", seguía activo y se pasaba por alto. Sería deseable usar el mismo plan con elementos de tamaño variable, pero ahora existe un problema para determinar los límites entre elementos. ¿Dónde termina un elemento y comienza el siguiente? Sin esta información no es posible recolectar la basura.

La solución más simple consiste en mantener, junto con el bit de recolección de basura en la primera palabra de cada bloque, activo o no, un *indicador de longitud* entero que especifique la longitud del bloque. Con los indicadores explícitos de longitud presentes, es de nuevo posible realizar un examen secuencial escrutando sólo la primera palabra de cada bloque. Durante este examen, los bloques libres adyacentes también se pueden compactar en bloques únicos antes de devolverse a la lista de espacios libres, con lo que se elimina el problema de compactación parcial que se analiza más adelante.

La recolección de basura también se puede combinar eficazmente con la compactación cabal para eliminar del todo la necesidad de una lista de espacios libres. En este caso, sólo se necesita un simple apuntador de montículo.

Compactación y el problema de fragmentación de memoria

El problema que encara cualquier sistema de gestión de almacenamiento en montículos que usa elementos de tamaño variable es el de la *fragmentación* de memoria. Se comienza con un solo bloque grande de espacio libre. Conforme el cómputo avanza, este bloque se fragmenta de manera gradual en trozos más pequeños a través de la asignación, recuperación y nuevo uso. Si sólo se utiliza la técnica simple de asignación de primer ajuste o de mejor ajuste, es evidente que los bloques de espacio libre continúan dividiéndose en pedazos cada vez más pequeños. Finalmente, se alcanza un punto donde el asignador de almacenamiento no puede satisfacer una solicitud de un bloque de N palabras porque no existe un bloque suficientemente grande, no obstante que la lista de espacios libres contiene en total mucho más de N palabras. Sin cierta compactación de bloques libres a bloques más grandes, la ejecución se detendrá por falta de almacenamiento libre más pronto de lo necesario.

Es posible uno de dos enfoques para la compactación, de acuerdo con el hecho de si se puede desplazar la posición de bloques activos dentro del montículo:

1. *Compactación parcial.* Si los bloques activos *no se pueden* desplazar (o si es demasiado costoso hacerlo), entonces sólo se pueden compactar bloques libres adyacentes en la lista de espacios libres.
2. *Compactación cabal.* Si los bloques activos *se pueden* desplazar, entonces todos ellos se pueden trasladar a un extremo del montículo para dejar todo el espacio libre en el otro en un bloque contiguo. La compactación cabal exige que, cuando se desplaza un bloque activo, todos los apuntadores a ese bloque se modifiquen para que apunten a la nueva localidad.

Capítulo 6

Control de secuencia

En un lenguaje de programación, las estructuras de control proporcionan el marco básico dentro del cual las operaciones y datos se combinan en programas y conjuntos de programas. Hasta aquí nos hemos ocupado de datos y operaciones aisladas; ahora debemos considerar su organización en programas completos ejecutables. Esto implica dos aspectos: el control del orden de ejecución de las operaciones, tanto primitivas como definidas por el usuario, el cual designamos como *control de secuencia* y analizamos en este capítulo, y el control de la transmisión de datos entre los subprogramas de un programa, al cual llamamos *control de datos* y estudiamos en los dos capítulos siguientes. Esta división es conveniente porque ambos asuntos son más bien complejos, aunque también sirve para diferenciar claramente dos aspectos de los lenguajes de programación que a menudo se confunden.

6.1 CONTROL IMPLÍCITO Y EXPLÍCITO DE SECUENCIA

Las estructuras de control de secuencia se pueden clasificar convenientemente en tres grupos:

1. Estructuras que se usan en *expresiones* (y por tanto dentro de enunciados, puesto que las expresiones constituyen los bloques de construcción básicos para enunciados), como reglas de precedencia y paréntesis;

2. Estructuras que se usan entre *enunciados* o grupos de enunciados, como enunciados condicionales e iterativos; y

3. Estructuras que se usan entre *subprogramas*, como llamadas de subprograma y corrutinas. Las llamadas simples se estudiarán en este capítulo; las corrutinas y demás comunicación entre subprogramas se expondrán más a fondo en los capítulos 7 y 8.

Esta división es por necesidad algo imprecisa. Por ejemplo, ciertos lenguajes, como LISP y APL, no tienen enunciados, sólo expresiones, sin embargo aún se emplean versiones de los mecanismos usuales de control de secuencia de enunciados.

Las estructuras de control de secuencia pueden ser implícitas o explícitas. Las estructuras de control de secuencia *implícitas* (o por omisión) son las que el lenguaje define que están en

operación, a menos que el programador las modifique a través de alguna estructura explícita. Por ejemplo, casi todos los lenguajes definen que el orden físico de los enunciados de un programa controla el orden en el cual los enunciados se ejecutan, a menos que sea modificado por un enunciado de control de secuencia explícito. Dentro de las expresiones también hay comúnmente una jerarquía de operaciones definida por el lenguaje y que controla el orden de ejecución de las operaciones de la expresión cuando no hay paréntesis. Las estructuras *explícitas* de control de secuencia son aquellas que el programador puede usar en forma optativa para modificar el orden implícito de las operaciones definido por el lenguaje, como, por ejemplo, usando paréntesis dentro de las expresiones, o enunciados **goto** y etiquetas de enunciado.

6.2 SECUENCIAMIENTO CON EXPRESIONES ARITMÉTICAS

Considérese la fórmula para el cómputo de raíces de la ecuación cuadrática:

$$raíz = \frac{-B \pm \sqrt{B^2 - 4 \times A \times C}}{2 \times A}$$

Esta fórmula en apariencia sencilla implica en realidad al menos 15 operaciones individuales (suponiendo una primitiva de raíz cuadrada y contando las diversas referencias de datos). Codificada en un lenguaje ensamblador o de máquina típico, requeriría al menos 15 instrucciones, y probablemente muchas más. Más aún, el programador tendría que proporcionar almacenamiento para cada uno de los diversos resultados intermedios generados y seguir la pista de los mismos, además de que tendría que preocuparse por la optimización. ¿Se pueden combinar las referencias al valor de B y de A?, ¿en qué orden se deben efectuar las operaciones para minimizar el almacenamiento temporal y hacer el mejor uso del hardware?, etc. Sin embargo, en un lenguaje de alto nivel como FORTRAN, la fórmula para una de las raíces se puede codificar casi directamente como una sola expresión:

```
ROOT = (-B+SQRT(B**2-4*A*C)) / (2*A)
```

La notación es compacta y natural, y el procesador de lenguajes, en vez del programador, se ocupa del almacenamiento temporal y la optimización. Parece justo afirmar que la disponibilidad de expresiones en los lenguajes de alto nivel es una de sus ventajas principales sobre los lenguajes de máquina y ensamblador.

Las expresiones son un dispositivo poderoso y natural para representar series de operaciones, aunque plantean problemas nuevos. Si bien puede ser tedioso escribir largas series de instrucciones en lenguaje de máquina, al menos el programador entiende con claridad el orden en el que las operaciones se van a ejecutar. Pero, ¿qué hay de la expresión? Tómese la expresión en FORTRAN para la fórmula cuadrática. ¿Es correcta? ¿Cómo sabemos, por ejemplo, que la expresión indica que la sustracción debe tener lugar *después* del cómputo de $4 * A * C$ y no antes? Los mecanismos de control de secuencia que operan para determinar el orden de las operaciones dentro de esta expresión son, de hecho, bastante complejos y sutiles.

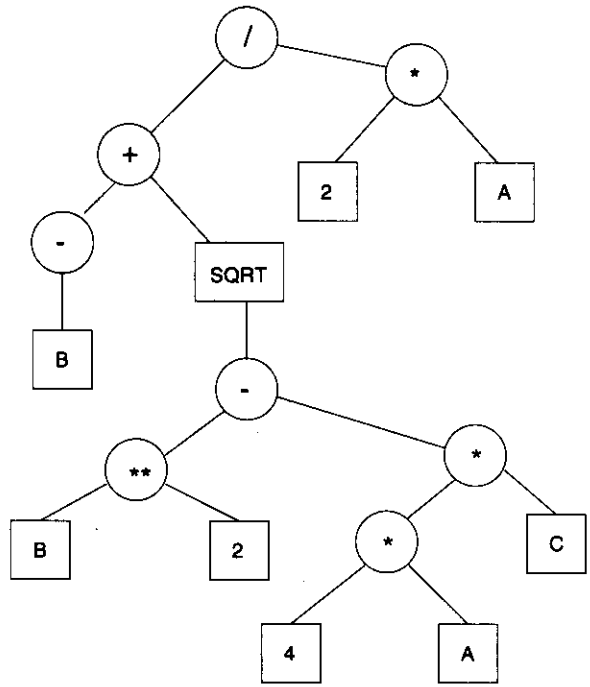


Figura 6.1. Representación de árbol de la fórmula cuadrática.

6.2.1 Representación de estructura de árbol

Hasta aquí, se ha considerado a las expresiones como entidades únicas y se ha pasado por alto la sintaxis y semántica reales necesarias para la evaluación de una expresión dada. Al considerar las operaciones dentro de expresiones, llamaremos *operandos* a los argumentos de una operación.

El mecanismo básico de control de secuencia en expresiones es la *composición funcional*: se especifica una operación y sus operandos; los operandos pueden ser constantes, objetos de datos u otras operaciones, cuyos operandos, a su vez, pueden ser constantes, objetos de datos o aun otras operaciones, a cualquier profundidad. La composición funcional confiere a una expresión la estructura característica de un árbol, donde el nodo raíz del árbol representa la operación principal, los nodos entre la raíz y las hojas representan operaciones de nivel intermedio, y las hojas representan referencias de datos (o constantes). Por ejemplo, la expresión para la fórmula cuadrática se puede representar (usando _ para simbolizar la operación menos unaria) mediante el árbol de la figura 6.1.

La representación de árbol aclara la estructura de control de la expresión. Es claro que los resultados de referencias de datos u operaciones a niveles más bajos en el árbol sirven como operandos para operaciones a niveles más altos en el mismo y, por tanto, estas referencias de datos y operaciones se deben evaluar (ejecutar) primero. Sin embargo, la representación

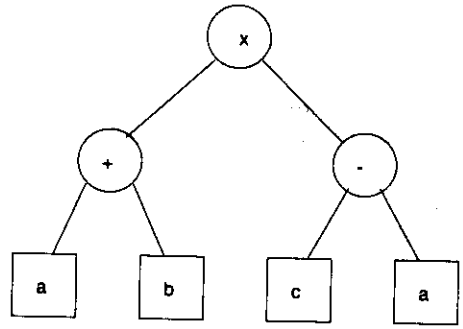


Figura 6.2. Forma de árbol para la expresión simple: $(a + b) \times (c - a)$.

de árbol deja parte del orden de evaluación sin definir. Por ejemplo, en el árbol de la figura 6.1 no está claro si $-B$ se debe evaluar antes o después de $B * *2$, ni tampoco si las dos referencias de datos al identificador B se pueden combinar en una sola referencia. Desafortunadamente, en presencia de operaciones con efectos colaterales puede significar una diferencia, como se verá más adelante. En una definición de lenguaje, es común definir el orden de evaluación de las expresiones sólo al nivel de la representación de árbol y permitir que el implementador del lenguaje decida acerca del orden de evaluación en detalle (por ejemplo, si es primero B o $B * *2$). Sin embargo, antes de examinar los problemas que surgen al determinar el orden exacto de evaluación, es apropiado dar un vistazo a las diversas representaciones sintácticas para expresiones que están en uso.

Sintaxis para expresiones

Si se toman las expresiones como representadas característicamente por árboles, entonces, para usar expresiones dentro de programas, se requiere cierta linealización de los árboles; es decir, se debe disponer de una notación para escribir árboles como series lineales de símbolos. Examinemos las notaciones más comunes:

Notación prefija (polaca prefija). Al escribir llamadas de función, se escribe ordinariamente el nombre de la función antes de sus argumentos, como en $f(x, y, z)$. Esto se puede hacer extensivo a todas las operaciones de una expresión. En notación por *prefijos*, se escribe primero el símbolo de la operación seguido de los operandos en orden de izquierda a derecha. Si un operando es a su vez una operación con operandos, se aplican entonces las mismas reglas. El árbol de la figura 6.2 se convierte así en $\times + a\ b - c\ a$. Puesto que + es un *operador diádico* (requiere dos argumentos), es claro que los argumentos para + son a y b. En forma similar, los argumentos para —deben ser c y a. Por último, los argumentos para × deben ser, por tanto, el término + y el término—. No hay ambigüedad alguna y no se necesitaron paréntesis para especificar exactamente cómo evaluar la expresión. Debido a que el matemático polaco Lukasiewicz inventó la notación libre de paréntesis, se ha aplicado el término *polaca* a esta notación y sus derivados.

Una variante de esta notación, que se usa en LISP, se designa a veces como *polaca de Cambridge*. En ésta, un operador y sus argumentos están rodeados por paréntesis. Una expresión se ve, por tanto, como un conjunto anidado de listas, donde cada lista comienza con un símbolo de operador seguido por las listas que representan los operandos. En polaca de Cambridge, el árbol de la figura 6.2 se convierte en $(\times\ (+ab)(-ca))$.

Considérese la fórmula cuadrática (figura 6.1) representada en notación prefija (usando $\uparrow$ para exponenciación, $\surd$ para *SQRT*, y _ para menos unaria):

$/ + \_\ b_{\surd} - \uparrow b\ 2 \times \times 4\ a\ c \times 2\ a$ (*polaca*)

$(/\ (+\ (\_b)(_{\surd}(-\ (\uparrow\ b\ 2)(\times\ (\times 4\ a)\ c))))(\times\ 2\ a))$ (*polaca de Cambridge*)

Notación posfija (sufija o polaca inversa). La notación posfija es similar a la notación prefija excepto que el símbolo de operación *sigue* a la lista de operandos. Por ejemplo, la expresión de la figura 6.2 se representa como $a\ b + c\ a - \times$.

Notación infija. La notación infija es más adecuada para operaciones binarias (diádicas). En esta notación, el símbolo de operador se escribe entre los dos operandos. Puesto que la notación infija para las operaciones aritméticas, relacionales y lógicas básicas es de uso tan común en las matemáticas ordinarias, la notación para estas operaciones ha sido adoptada ampliamente en lenguajes de programación y en ciertos casos se ha hecho también extensiva a otras operaciones. La forma infija del árbol de la figura 6.2 se representa como $(a + b) \times (c - a)$. Para operadores con más de dos argumentos (por ejemplo, la operación condicional en C), se usa infija de una manera algo torpe, empleando múltiples operadores infijos: $(expr?s_1 : s_2)$.

Semántica para expresiones

Cada una de estas tres notaciones, prefija, posfija e infija, tiene ciertos atributos que son útiles en el diseño de lenguajes de programación. Difieren principalmente en cuanto a la manera de calcular el valor para cada expresión. En lo que sigue proporcionamos algoritmos para evaluar (es decir, computar la semántica) para cada formato de expresión. Luego mostramos que los traductores tienen la opción de usar este proceso o hacerle modificaciones menores con el fin de hacer más eficiente la evaluación de expresiones.

Evaluación prefija. Con notación prefija, se puede evaluar cada expresión en un solo examen de la misma. Es necesario conocer, sin embargo, el número de argumentos para cada operación. Es por esta razón que se necesitan símbolos especiales para la sustracción diádica (–) y la menos unaria (–) para distinguir cuál es la operación deseada (o bien usar polaca de Cambridge con paréntesis).

Además del ahorro de paréntesis, la notación prefija tiene cierto valor en el diseño de lenguajes de programación:

1. Como ya se ha señalado, la llamada de función usual ya está escrita en notación prefija.

2. La notación prefija se puede usar para representar operaciones con cualquier número de operandos y es, por tanto, completamente general. Sólo es necesario aprender una regla sintáctica para escribir cualquier expresión. Por ejemplo, en LISP, sólo se requiere dominar la notación polaca de Cambridge para escribir cualquier expresión, y con ello se habrán aprendido casi todas las reglas sintácticas del lenguaje.

3. También es relativamente fácil decodificar mecánicamente la notación prefija; por esta razón, se consigue sin dificultad la traducción de expresiones prefijas a secuencias de código simples.

Este último punto —de fácil traducción dentro de secuencias de código— se puede demostrar por medio del algoritmo siguiente. Dada la expresión prefija *P* compuesta de operadores y operandos, se puede evaluar la expresión con una pila de ejecución:

1. Si el elemento siguiente en *P* es un operador, agregarlo en el tope de la pila. Fijar la cuenta de argumentos en el número de operandos que necesita el operador. (Si el número es *n*, se dice que el operador es un operador *n*-ario).

2. Si el próximo elemento en *P* es un operando, agregarlo en el tope de la pila.

3. Si las *n* entradas superiores de la pila son entradas de operando necesarias para el operador *n*-ario superior (por ejemplo, si + era el último operador en la pila y se agregaron dos operandos a la misma), se puede aplicar el operador superior a estos operandos. Reemplazar el operador y sus *n* operandos por el resultado de aplicar esa operación a los *n* operandos.

Si bien esto es bastante sencillo, se tiene el problema de que, después de agregar un operando en la pila, todavía es necesario verificar si se dispone de suficientes operandos para satisfacer el operador superior actual. El uso de notación posfija evita eso.

Evaluación posfija. Puesto que en la notación posfija el operador sigue a sus operandos, cuando se examina un operador sus operandos ya han sido evaluados. Por consiguiente, la evaluación de una expresión posfija *P*, usando una vez más una pila, ahora tiene lugar como sigue:

1. Si el elemento siguiente de *P* es un operando, colocarlo en la pila.

2. Si el elemento siguiente de *P* es un operador *n*-ario, entonces sus *n* argumentos *deben* ser los *n* elementos superiores en la pila. Reemplazar estos *n* elementos por el resultado de aplicar esta operación usando los *n* elementos como argumentos.

Como se ha visto, la estrategia de evaluación es directa y fácil de implementar. De hecho, es la base del código generador de expresiones en muchos traductores. Durante la traducción (véase el capítulo 3), la sintaxis de las expresiones a menudo se vuelve posfija.

El generador del código usará el algoritmo antes mencionado para determinar el orden del código de generación para contar el valor de la expresión.

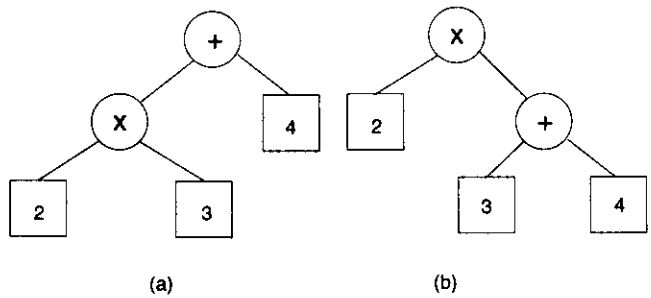


Figura 6.3. Orden de evaluación de operadores.

Evaluación infija. Aunque la notación infija es común, su uso en un lenguaje de programación conduce a varios problemas singulares:

1. Puesto que la notación infija es adecuada sólo para operadores binarios, un lenguaje no puede usar sólo notación infija, sino que debe combinar por necesidad notaciones infija y prefija (o posfija). La mezcla hace que la traducción sea consecuentemente más compleja. Los operadores unarios y las llamadas de función con argumentos múltiples deben ser excepciones a la propiedad infija general.

2. Cuando aparece más de un operador infijo en una expresión, la notación es inherentemente ambigua a menos que se utilicen paréntesis.

Este último punto se puede demostrar considerando el valor para la expresión infija $2 \times 3 + 4$. Sin duda, usted entendió que esta expresión significa el valor 10, pero con la misma facilidad podría haber representado el valor 14. La figura 6.3 muestra por qué. Cuando usted aprendió a sumar y a multiplicar, también aprendió la regla: "hacer la multiplicación antes que la suma" [figura 6.3(a)]. Sin embargo, eso es simplemente una convención, y las ma-temáticas se podrían haber desarrollado de igual manera con un supuesto alternativo de efectuar la suma antes que la multiplicación [figura 6.3(b)]. Siempre se pueden usar paréntesis para eliminar la ambigüedad de cualquier expresión indicando explícitamente el agrupamiento de operadores y operandos, como en $(a \times b) + c$ o $a \times (b + c)$, pero en expresiones complejas los nidos profundos de paréntesis que resultan se vuelven confusos.

Por esta razón, los lenguajes introducen por lo común reglas implícitas de control que hacen innecesarios casi todos los usos de los paréntesis. Se describen dos ejemplos de estas reglas implícitas.

Jerarquía de operaciones (por ejemplo, *reglas de precedencia*). Los operadores que pueda haber en las expresiones están colocados en una jerarquía u orden de precedencia. La jerarquía de Ada es representativa (véase la tabla 6.1). En una expresión donde intervienen operadores de más de un nivel en la jerarquía, la regla implícita es que los operadores con mayor precedencia se deben ejecutar primero. Así, en $a \times b + c$, $\times$ está arriba de $+$ en la jerarquía y se evaluará primero.

Nivel de precedencia	Operadores	Operaciones
Precedencia más alta	$\star\star$ *abs not*	Exp, valor abs., negación
	$\star$ / *mod rem*	Multiplicación, división
	$+ -$	Adición, sustracción unarias
	$+ - \&$	Adición, sustracción binarias
	$= \leq < > \geq$	Relacionales
Precedencia más baja	*and or xor*	Operaciones booleanas

Tabla 6.1. Jerarquía de operaciones en Ada.

Precedencia	Operadores	Nombres de operadores
17	componentes léxicos, a[k], f()	Literales, subindización, llamada de función
	., −>	Selección
16	++, −−	Incremento/decremento posfijo
15*	++, −−	Inc./dec. prefijo
	~, -, sizeof	Operaciones unarias, almacenamiento
	!, &, *	Negación lógica, indirección
14	(*typename*)	Conversiones forzadas
13	*, /, %	Operadores multiplicativos
12	+, −	Operadores aditivos
11	<<, >>	De desplazamiento
10	<, >, <=, >=	Relacionales
9	==, !=	De igualdad
8	&	and por bits
7	∧	xor por bits
6	|	or por bits
5	&&	and lógico
4	||	o lógico
3*	? :	Condicionales
2*	=, + =, − =, * =, / =, % =, <<=, >>=, & =, ∧ =, | =	De asignación
1	,	Evaluación secuencial

* - Operaciones asociativas derechas

Tabla 6.2. Niveles de precedencia en C para operadores.

Asociatividad. En una expresión donde intervienen operaciones del mismo nivel en la jerarquía, se necesita una regla implícita adicional de asociatividad para definir cabalmente el orden de las operaciones. Por ejemplo, en $a - b - c$, ¿se debe evaluar en primer término la primera o la segunda sustracción? La asociatividad de izquierda a derecha es la regla implícita más común, de modo que $a - b - c$ se trata como $(a - b) - c$. [Sin embargo, las

convenciones matemáticas comunes dicen que la exponenciación opera de derecha a izquierda: $a \uparrow b \uparrow c = a \uparrow (b \uparrow c)$.]

La precedencia funciona bastante bien para las expresiones aritméticas usuales, puesto que el modelo matemático subyacente de semántica de expresiones es bien conocido para casi todos los programadores. Sin embargo, conforme los lenguajes evolucionan con nuevos operadores que no son de las matemáticas clásicas, las precedencias dejan de funcionar. C, APL, Smalltalk y Forth son todos ejemplos de lenguajes que manejan conjuntos ampliados de operadores en diferentes formas:

- *C.* C utiliza un conjunto ampliado de tablas de precedencia, como se muestra en la tabla 6.2. Casi todas las entradas emplean asociatividad de izquierda a derecha, excepto las marcadas con una estrella. Casi todos los niveles de precedencia son razonables. Pascal, por otra parte, tiene una extraña anomalía en su tabla de precedencia. La expresión booleana $a = b \mid c = d$ es ilegal, pues la precedencia del Pascal supone el agrupamiento no natural de $a = (b|c) = d$, que no es lo que el programador quiere decir ordinariamente. En Pascal, es más seguro usar siempre paréntesis al escribir expresiones lógicas.

Sinopsis de lenguaje 6.1: APL

Características: El procesamiento de arreglos es el punto fuerte del APL. Todas las operaciones aritméticas son aplicables a vectores y arreglos, y existen operaciones adicionales para crear vectores especiales, por ejemplo, todos los elementos fijados en valores dados. Puesto que muchas de estas operaciones vectoriales no son intuitivas, no existe precedencia alguna en APL y los enunciados se ejecutan de derecha a izquierda —una peculiaridad extraña hasta que uno se acostumbra a ella—.

Historia: El APL fue desarrollado por Ken Iverson a principios de los años sesenta como una notación para describir computación. Más tarde se usó como un lenguaje arquitectónico de máquina donde el comportamiento de las instrucciones individuales se podía describir con facilidad como operaciones vectoriales en APL. Una tercera fase en el desarrollo del APL fue una implementación para la IBM 360 a finales de los años sesenta. A causa de lo conciso de las expresiones en APL, este lenguaje desarrolló un número reducido pero dedicado de seguidores que se enorgullecen en desarrollar programas complejos.

Ejemplo: Sumar elementos de un arreglo

1 k ← □	Leer tamaño del arreglo en k
2 A ← □	Leer el primer elemento de un arreglo
3 A ← A, □	Leer próximo elemento del arreglo. Expandir arreglo
4 → 3 × *i* K > ρ A	Tamaño de A (ρ A) comparado con k dando 0 o 1. Generar vector de 0 o 1 (operador *i*) y goto (→) enunciado 3 si el resultado es 1 (es decir, no hecho todavía)
5 □ ← +/A	Sumar elementos del arreglo (+/) y producir suma

Referencia: A. D. Falkoff y K. E. Iverson, "The evolution of APL," *ACM History of Programming Languages Conference*, Los Ángeles, CA (Junio 1978) (*SIGPLAN Notices(13)8 [Agosto 1978])*, 47-57.

- *APL.* El APL es un lenguaje cuyos operadores primitivos están proyectados para actuar sobre arreglos y vectores. Casi todos los operadores de APL son "nuevos" en el sentido de que los programadores que no usan APL los encuentran extraños al estudiarlos por primera vez. Cualquier asignación de precedencia a operadores de APL sería artificial, de modo que el lenguaje está proyectado sin precedencias y todas las expresiones se evalúan de derecha a izquierda. Esto es razonable en casi todos los programas en APL, excepto que ciertas expresiones "típicas" se hacen no intuitivas. Por ejemplo, $a - b - c$, donde a, b y c son enteros, se evalúa como $a - (b - c)$, que es lo mismo que $a - b + c$ en lenguajes como Pascal, FORTRAN o C. El APL se resume en la sinopsis de lenguaje 6.1.

- *Smalltalk.* Este lenguaje emplea un modelo como el APL. Puesto que el objetivo en el caso de Smalltalk es desarrollar funciones (*métodos*) que proporcionen la funcionalidad requerida, nunca queda claro qué precedencia debe tener una función nueva. Por consiguiente, la precedencia se omite por lo general y la evaluación de expresiones tiene lugar de izquierda a derecha. (Hay más información sobre Smalltalk en la sección 8.2.4.)

- *Forth.* Forth era un lenguaje proyectado para operar en computadoras de proceso en tiempo real. Éstas se desarrollaron en la década de 1960 como computadoras pequeñas de bajo costo. La memoria era costosa. Cualquier lenguaje de programación tenía que ser pequeño, fácil de traducir y eficiente en cuanto a ejecución. Como ya se ha indicado, la evaluación de expresiones posfijas se puede efectuar con facilidad. Forth era un lenguaje cuya estructura en tiempo de ejecución era una pila, y su sintaxis era posfija pura. Esto permitió instalar fácilmente traductores a un costo bajo en muchas aplicaciones. Con posfija pura, la precedencia no representaba ya un problema. El Forth se resume en la sinopsis de lenguaje 6.2.

Cada una de las notaciones para expresiones que hemos mencionado tiene sus propias dificultades particulares. La notación infija con la precedencia y reglas de asociatividad implícitas y uso explícito de paréntesis (cuando se requiere) proporciona una representación bastante natural para casi todas la expresiones aritméticas, relacionales y lógicas. Sin embargo, la necesidad de las reglas implícitas complejas y el uso obligado de notación prefija (u otra) para operaciones no binarias hace que la traducción de estas expresiones sea compleja. La notación infija sin las reglas implícitas (es decir, con pleno uso de paréntesis) es engorrosa a causa del gran número de paréntesis necesarios. Sin embargo, tanto la polaca de Cambridge como la notación matemática prefija ordinaria comparten este problema con los paréntesis. La notación polaca evita por completo el uso de paréntesis, pero es necesario conocer por adelantado el número de operandos que se requieren para cada operador, una condición que suele ser difícil de satisfacer cuando intervienen operaciones definidas por el programador. Además, la falta de claves de estructuración dificulta la lectura de expresiones polacas complejas. Todas las notaciones prefijas y posfijas comparten la ventaja de aplicarse a operaciones con diferentes números de operandos.

Sinopsis de lenguaje 6.2: Forth

Características: Lenguaje fuente posfijo que conduce a un modelo de ejecución eficiente, aun cuando por lo general se interpreta. El sistema se ejecuta en dos pilas: una pila de retorno de subrutinas y una pila de evaluación de expresiones. Modelo en tiempo de ejecución muy pequeño, lo cual lo hace útil en computadoras pequeñas incrustadas.

Historia: El Forth fue desarrollado por Charles Moore alrededor de 1970. El nombre es una contracción de "Fourth Generation Programming Language" limitada a cinco caracteres. El lenguaje fue un sustituto de FORTRAN en computadoras pequeñas en los años setenta, cuando el espacio era valioso y el único dispositivo de entrada/salida solía ser una muy lenta y engorrosa cinta de papel. El hecho de contar con un traductor/intérprete residente facilitaba el desarrollo de programas en el sistema objetivo.

Ejemplo: Programa para el cómputo de: $1^2 + 2^2 + \ldots + 9^2 + 10^2$

```
                        (Notación: a, b, c es pila de expresiones, c es pila(tope))
:SQR DUP*;              (Define cuadrado por: n ⇒ n, n ⇒ (n * n))
:DOSUM SWAP 1 + SWAP OVER SQR +;   (N, S ⇒ N + 1, S + (N + 1)²)
                        (N, S ⇒ S, N ⇒ S,(N + 1) ⇒ (N + 1),S ⇒
                        (N + 1), S,(N + 1) ⇒ (N +1),S,(N + 1)² ⇒
                        (N + 1), S + (N + 1)²)
3 6 DOSUM . . 22 4 ok   (Punto (.) imprime pila(tope). Salida es 22 = 4² + 6)
0 0 10 0 DOSUM LOOP .385 ok   (Aplicar DOSUM de 0 a 9 (Alto en 10))
```

Referencia: E. Rather, D. Colburn y C. Moore, "The evolution of Forth," *ACM History of Programming Languages Conference II*, Cambridge, MA (Abril 1993) *(SIGPLAN Notices (28)3 [Marzo 1993])*, 177-199.

6.2.2 Representación en tiempo de ejecución

Anteriormente se proporcionaron algoritmos para entender la semántica de expresiones escritas en cada una de las tres formas comunes. Sin embargo, si primero se transforma cada expresión en su representación de árbol, se permite al traductor hacer elecciones alternativas para la evaluación eficiente de la expresión. La primera etapa de esta traducción establece la estructura básica de control de árbol de la expresión utilizando las reglas implícitas de precedencia y asociatividad cuando en la expresión interviene notación infija. En una segunda etapa optativa, se toman las decisiones en detalle respecto al orden de evaluación, lo cual incluye la optimización del proceso de evaluación.

A causa de la dificultad para decodificar expresiones en su forma original infija en el texto del programa, es común traducirlas a una forma ejecutable que se puede decodificar con facilidad durante la ejecución. Las siguientes son las alternativas más importantes en uso:

1. *Secuencias de código de máquina.* La expresión se traduce a código de máquina real efectuando las dos etapas de traducción indicadas en un paso. El ordenamiento de las

instrucciones refleja la estructura de control de secuencia de la expresión original. En computadoras convencionales, estas secuencias de código de máquina deben hacer uso de localidades explícitas de almacenamiento temporal para guardar resultados intermedios. La representación en código de máquina, desde luego, permite el uso del intérprete de hardware, lo que proporciona una ejecución muy rápida.

2. *Estructuras de árbol.* Las expresiones se pueden ejecutar directamente en su representación natural de estructura de árbol (primera etapa) usando un intérprete de software. La ejecución (segunda etapa) se puede conseguir entonces a través de un simple recorrido de árbol. Ésta es la técnica básica que se usa en LISP (interpretado por software), donde se representan programas completos como estructuras de árbol durante la ejecución.

3. *Forma prefija o posfija.* Las expresiones en forma prefija o posfija (ambas etapas en un solo paso) pueden ser ejecutadas por el sencillo algoritmo de interpretación que se ha dado antes. En ciertas computadoras reales con base en una organización de pila, el código de máquina real está representado esencialmente en forma posfija. La representación prefija es la forma ejecutable de programas en SNOBOL4 en muchas implementaciones. La ejecución es por examen de izquierda a derecha, y cada operación llama al intérprete en forma recursiva para evaluar sus operandos.

Evaluación de representaciones de árbol de expresiones

Aunque la traducción a partir de expresiones de programas a representaciones de árbol causa ocasionalmente dificultades, el procedimiento básico de traducción es sencillo. La segunda etapa, en la cual el árbol se traduce a una serie ejecutable de operaciones primitivas, implica la mayoría de las cuestiones sutiles de orden de evaluación. No es nuestro interés estudiar aquí algoritmos para la generación de código ejecutable a partir de la representación de árbol, sino más bien considerar los problemas de orden de evaluación que surgen al determinar con exactitud el código por generar.

Problema 1. Reglas de evaluación uniforme. Al evaluar una expresión, o al generar código para su evaluación, uno esperaría que se aplicara la regla de evaluación uniforme siguiente: para cada nodo de operaciones en el árbol de expresión, evalúese (o genérese código para evaluar) primero cada uno de sus operandos, y luego aplíquese la operación (o genérese código para aplicar la operación) a los operandos evaluados. A esto le llamamos la regla de evaluación *impaciente* porque siempre se evalúan primero los operandos. El orden exacto en el que estos operandos se presentan no debe importar, así que se puede elegir el orden de evaluación de operandos o de operaciones independientes para optimizar el uso de almacenamiento temporal u otras características de la máquina. Bajo esta regla de evaluación, para la expresión $(a + b) \times (c - a)$ de la figura 6.2 sería aceptable cualquiera de los órdenes de evaluación siguientes:

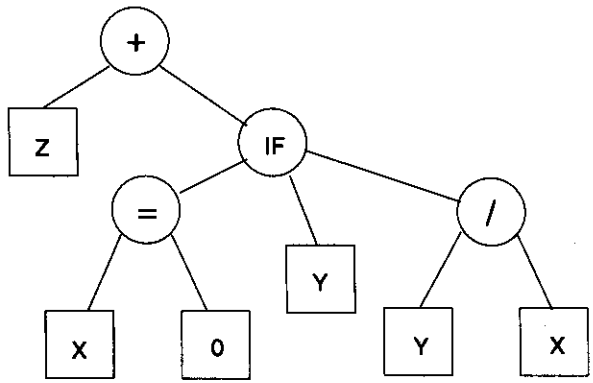


Figura 6.4. Expresión que contiene una condicional.

Orden 1. Calcúlese primero $a + b$:

1. Obténgase el valor r de a.
2. Obténgase el valor r de b.
3. Súmese a y b para obtener d.
4. Obténgase el valor r de c.
5. Réstese a de c para obtener e.
6. Multiplíquese d y e para obtener f, el valor r de la expresión.

Orden 2. Evalúense los operandos antes que cualquier operador:

1. Obténgase el valor r de c.
2. Obténgase el valor r de b.
3. Obténgase el valor r de a.
4. Réstese a de c para obtener e.
5. Súmese a y b para obtener d.
6. Multiplíquese d y e para obtener f.

Todo esto es bastante natural, y sería deseable suponer siempre esta regla de evaluación uniforme. Por desgracia, no siempre se aplica. El mejor ejemplo es el caso de expresiones que contienen condicionales, por ejemplo, en C la expresión $Z + (Y = 0\ ?\ X : X/Y)$ tiene un **if** incrustado que calcula – si Y no es 0. Sería deseable tratar una condicional como ésta simplemente como una operación con una "sintaxis rara" y tres operandos, como en la figura 6.4. De hecho, esto es exactamente lo que se hace en LISP, utilizando la notación polaca de Cambridge para condicionales lo mismo que para todas las demás operaciones. Pero ahora se tiene un problema con la regla de evaluación uniforme. Si se supone la regla y se evalúan los operandos del operador condicional de la figura 6.4, se produce el efecto de hacer exactamente lo que se busca evitar con la condicional, esto es, dividir X entre Y incluso si Y es cero. Es

claro que en este caso no se desea que todos los operandos se evalúen antes de que se aplique la operación. En vez de ello, es necesario pasar los operandos (o al menos los dos últimos operandos) a la operación condicional *sin evaluar* y dejar que la operación determine el orden de evaluación.

El problema con las condicionales sugiere que quizá sería mejor una regla alternativa de evaluación uniforme, a la que se suele llamar regla de evaluación *perezosa*: *nunca* evalúe operandos antes de aplicar la operación; en vez de ello, siempre pase los operandos sin evaluar y deje que la operación decida si la evaluación es necesaria. Esta regla de operación funciona en todos los casos y, por tanto, en teoría, debe servir. Sin embargo, la implementación resulta impráctica en muchos casos porque, ¿cómo se va a simular el paso de operandos sin evaluar a operaciones? Esto requiere considerable simulación de software para lograrse. Los lenguajes interpretativos como LISP y Prolog suelen emplear este enfoque para la evaluación de expresiones, pero para los lenguajes parecidos a la aritmética, como C y FORTRAN, el costo indirecto de manejar una evaluación perezosa de este tipo sería prohibitivo.

Las dos reglas de evaluación uniforme recién sugeridas, la *impaciente* y la *perezosa*, corresponden a dos técnicas comunes para pasar parámetros a subprogramas, transmisión *por valor* y *por nombre*, respectivamente. Los detalles de estos conceptos y su simulación se analizan más a fondo al tratar la transmisión de parámetros en el siguiente capítulo. Para los propósitos inmediatos, basta con señalar que ninguna regla simple de evaluación uniforme (o para generar código para expresiones) es satisfactoria. En implementaciones de lenguajes, es común encontrar una mezcla de las dos técnicas. En LISP, por ejemplo, las funciones (operaciones) se dividen en dos categorías de acuerdo con el hecho de si la función recibe operandos evaluados o sin evaluar. En SNOBOL4, las operaciones (subprogramas) definidas por el programador siempre reciben operandos evaluados, en tanto que las operaciones primitivas definidas por el lenguaje reciben operandos sin evaluar, con condicionales simuladas por secuencias de código en línea, pero los subprogramas definidos por el programador pueden recibir operandos tanto evaluados como sin evaluar.

Problema 2. Efectos colaterales. El uso de operaciones que tienen efectos colaterales en las expresiones es la base de una controversia ya antigua en el diseño de lenguajes de programación. Considérese la expresión:

$$a \times fun(x) + a$$

Antes de que se pueda efectuar la multiplicación, se debe obtener el valor r de a y se debe evaluar $fun(x)$. La suma requiere el valor de a y el resultado de la multiplicación. Es claramente deseable obtener el valor de a sólo una vez y simplemente usarlo en dos lugares en el calculo. Más aún, no debería haber diferencia alguna si $fun(x)$ se evalúa antes o después de la obtención del valor de a. Sin embargo, si *fun* tiene el efecto colateral de cambiar el valor de a, entonces el orden exacto de evaluación es crítico. Por ejemplo, si a tiene el valor inicial 1 y $fun(x)$ devuelve 3 y también cambia el valor de a a 2, entonces los valores posibles para esta expresión pueden ser:

1. *Evaluar cada término en orden:* $1 \times 3 + 2 = 5$

2. *Evaluar a sólo una vez:* $1 \times 3 + 1 = 4$
3. *Llamar fun(x) antes de evaluar a:* $3 \times 2 + 2 = 8$

Todos estos valores son "correctos" de acuerdo con la sintaxis del lenguaje y están determinados por el orden de ejecución de los componentes de la expresión.

Han surgido dos posturas respecto al uso de efectos colaterales en expresiones. Una postura es que los efectos colaterales se deben proscribir en las expresiones, ya sea no permitiendo en absoluto funciones con efectos colaterales o, simplemente, haciendo indefinido el valor de cualquier expresión en la que los efectos colaterales podrían afectar el valor (por ejemplo, el valor de la expresión anterior). Otro punto de vista es que los efectos colaterales se deben permitir y que la definición del lenguaje debe dejar claro exactamente cuál va a ser el orden de evaluación de una expresión para que el programador pueda hacer un uso adecuado de los efectos colaterales en el código. La dificultad con esta última postura es que hace imposible muchos tipos de optimización. En muchas definiciones de lenguaje la cuestión se pasa simplemente por alto, con el resultado infortunado de que diferentes implementaciones proporcionan interpretaciones en conflicto.

Ordinariamente, se permite que los enunciados tengan efectos colaterales. Por ejemplo, la operación de asignación produce necesariamente un efecto colateral: un cambio en el valor de una variable o elemento de estructura de datos. Y está claro que se espera que los efectos colaterales que produce un enunciado afecten las entradas del siguiente enunciado en la serie. El problema es si se debe permitir esta clase de interdependencia a través de efectos colaterales abajo del nivel de los enunciados, en expresiones. Si no se permite, es necesario especificar el orden de evaluación en las expresiones sólo hasta el nivel de representación de árbol; la evaluación de expresiones, para el programador, no tiene "trucos", y es posible la optimización de secuencias de evaluación de expresiones por parte del traductor. Sin embargo, si la optimización no es una preocupación primordial, suele ser provechoso permitir efectos colaterales y especificar cabalmente el orden de evaluación. En este caso, se pierde gran parte de la razón para distinguir entre enunciados y expresiones en un lenguaje. De hecho, en varios lenguajes, en particular LISP y APL, la distinción entre expresiones y enunciados ha desaparecido casi totalmente o por completo. Para el programador esto representa una simplificación valiosa. Así pues, no existe una postura dominante en cuanto a efectos colaterales en expresiones; ambos enfoques tienen partidarios.

Problema 3. Condiciones de error. Una clase especial de efecto colateral interviene en el caso de operaciones que pueden fallar y generar una condición de error. A diferencia de los efectos colaterales ordinarios, que por lo común se restringen a funciones definidas por el programador, pueden surgir condiciones de error en muchas operaciones primitivas (desbordamiento, dividir entre cero). Es indeseable proscribir efectos colaterales de esta clase; sin embargo, el significado, e incluso la incidencia de estas condiciones de error, pueden ser afectados por diferencias en el orden de evaluación de componentes de expresión. En tales situaciones el programador puede requerir un control preciso del orden de evaluación, pero la demanda de optimización puede impedirlo. La solución a estas dificultades tiende a ser en esencia *ad hoc*, y varía de lenguaje a lenguaje y de implementación a implementación.

Problema 4. Expresiones booleanas en cortocircuito. En programación suele ser natural usar las operaciones booleanas *and* (*o*) (&& en C) y *or* (*o*) (|| en C) para combinar expresiones relacionales como los enunciados en C:

```
if ((A == 0) || (B/A > C)) { ... }
while (( I <= UB) && (V[I] > C)) { ... }
```

En ambas expresiones la evaluación del segundo operando de la operación booleana puede conducir a una condición de error (división entre cero, error de ámbito de subíndices); el primer operando se incluye para asegurar que el error no ocurra. En C, si la expresión izquierda se evalúa como *cierto* en el primer ejemplo y *falso* en el segundo, entonces la segunda expresión nunca se evalúa. Lógicamente, este intento tiene sentido porque es claro que el valor de la expresión $\alpha \,||\, \beta$ es *cierto* si α es *cierto*, y de igual manera $\alpha \&\& \beta$ es *falso* si α solo es *falso*. Desafortunadamente, el problema de evaluación uniforme antes mencionado también está presente aquí: en muchos lenguajes ambos operandos se evalúan antes que la operación booleana. Muchos errores de programación provienen de la expectativa de que el valor del operando izquierdo de una operación booleana pueda "poner en cortocircuito" el resto de la evaluación si se puede decidir cuál es el valor de la expresión global a partir del valor del operando izquierdo solo. En Ada, una solución a este problema es incluir dos operaciones booleanas, *and then* (*y entonces*) y *or else* (*si no*), que proporcionan explícitamente evaluación en cortocircuito, además de las operaciones booleanas ordinarias *and* (*y*) y *or* (*o*), que no lo hacen. Por ejemplo, en Ada:

```
if (A = 0) or else ( B/A > C) then ...
```

no puede fallar debido a división entre cero, puesto que si A = 0 la evaluación de la expresión completa concluye y el valor se toma como *cierto*.

6.3 SECUENCIAMIENTO CON EXPRESIONES NO ARITMÉTICAS

En la sección anterior se analizó el orden de ejecución en la evaluación de expresiones aritméticas. Sin embargo, en los lenguajes están presentes otros formatos de expresiones. Los lenguajes como ML y Prolog, proyectados para procesar datos de caracteres, incluyen otras formas de evaluación de expresiones.

6.3.1 Concordancia de patrones

Una operación crucial en lenguajes como SNOBOL4, Prolog y ML es la concordancia de patrones. En este caso, la operación tiene éxito haciendo concordar y asignando un conjunto de variables a una plantilla predefinida. El reconocimiento de árboles de análisis sintáctico en gramáticas BNF en la sección 3.3.1 es representativo de esta operación.

Por ejemplo, la gramática siguiente reconoce palíndromos de longitud impar en todo el alfabeto 0 y 1:

$$A \rightarrow 0A0 \mid 1A1 \mid 0 \mid 1$$

El reconocimiento de la cadena válida 00100 tiene lugar así:

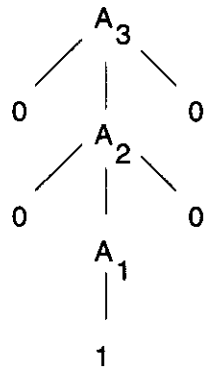


Figura 6.5. Concordancia de patrones en SNOBOL4.

A_1 *coincide con el centro* 1
A_2 coincide con $0A_10$
A_3 coincide con $0A_20$

A partir de estas tres "asignaciones" de A_1 a 1, A_2 a $0A_10$ y A_3 a $0A_20$, se puede construir el árbol de análisis sintáctico completo de esta cadena (figura 6.5).

SNOBOL4 es un lenguaje proyectado para simular esta característica directamente, como lo muestra la sinopsis de lenguaje 6.3. Obsérvese que el programa de ejemplo sólo tiene nueve enunciados de longitud; sería difícil reproducir eso en cualquier otro lenguaje existente en un conjunto tan breve y poderoso de enunciados.

SNOBOL4 también tiene otro atributo interesante. Su implementación se proyecta con independencia de cualquier arquitectura de máquina real. Está planeado para una máquina virtual de procesamiento de cadenas (como lo describe la figura 2.4 de la sección 2.2.3). Todo lo que se necesita es implementar las operaciones de máquina de procesamiento de cadenas como macros en una computadora existente para ejecutar SNOBOL4. Por esta razón, SNOBOL4 fue uno de los primeros lenguajes en (1) estar disponible en casi todas las computadoras, y (2) tener *exactamente la misma semántica* en todas las implementaciones.

En tanto que SNOBOL4 emplea reemplazo de cadenas para su operación de concordancia de patrones, Prolog utiliza el concepto de una relación como un conjunto de n-tuplas como mecanismo de concordancia. Por medio de la especificación de casos conocidos de estas relaciones (llamadas *hechos*), es posible deducir otros casos. Por ejemplo, se puede considerar la relación *PadreDe* con los hechos:

PadreDe(*Juan*,*María*).	- Juan es padre de María
PadreDe(*Susana*,*María*).	- Susana es padre de María
PadreDe(*Carlos*, *Juan*).	- Carlos es padre de Juan
PadreDe(*Ana*, *Juan*).	- Ana es padre de Juan

Para encontrar al padre de María se escribe simplemente la relación *PadreDe*(*X*,*María*) y Prolog tratará de asignar un valor a *X* a partir del conjunto conocido de hechos de su base de datos e inferirá que *X* puede ser ya sea Juan o Susana. Si se desea encontrar a ambos padres

Sinopsis de lenguaje 6.3: SNOBOL4

Características: Concordancia de patrones con base en gramáticas BNF; totalmente dinámico —incluso declaraciones, tipos, asignación de almacenamiento, hasta puntos de entrada y salida de procedimientos—; la implementación emplea macros virtuales de procesamiento de cadenas —reescritura de macros para cualquier computadora actual existente—.

Historia: El desarrollo fue iniciado en 1972 en los Laboratorios Bell de AT&T por Ralph Griswold, Ivan Polonsky y David Farber. La meta era desarrollar un lenguaje de procesamiento de cadenas para manipulación de fórmulas y análisis de gráficas. En los años cincuenta, Yngve, en el MIT, desarrolló COMIT como una forma de manejar procesamiento de lenguajes naturales usando reglas BNF, pero el equipo de los Laboratorios Bell sintió que era muy restrictivo para sus necesidades.

Llamado originalmente Symbolic Computation Language 7 (SCL7), su nombre cambió a SEXI (String Expression Interpreter), que no tenía futuro por razones obvias en los años sesenta, y luego a SNOBOL (StriNg Oriented symBOlic Language) como un acrónimo no intuitivo planeado intencionalmente. Se desarrollaron varias versiones de SNOBOL: SNOBOL, SNOBOL2, SNOBOL3 y SNOBOL4; este último tuvo éxito en los años setenta.

Ejemplo: Encontrar palíndromo más largo de longitud non sobre 0 y 1 en cadenas de entrada.

```
START   GRAMMAR = 0 | 1 | 0 *GRAMMAR 0 | 1 *GRAMMAR 1
*           Fijar patrón como gramática BNF
LOOP    NEWLINE = TRIM(INPUT)                       :F(END)
*           Obtener próxima línea sin espacios colgantes. Si fracaso, ir a END
        NEWLINE (POS(0) SPAN("01") RPOS(0))     :F(BAD)
*           Hacer coincidir renglón con 0s y 1s.
*           SPAN es cadena de 0s y 1s
*           POS(0) es primer pos. RPOS(0) es el último.
        SN = SIZE(NEWLINE)
NEXT    NEWLINE POS(0) GRAMMAR . PALINDROME POS(SN)
-               :S(OK) F(NOTOK)
*           Renglón coincide con gramática a través de POS(SN).
*           Si fracaso, mover último pos. Si éxito, imprimir respuesta.
*           Parte igualada asignada a PALINDROME
OK      OUTPUT = "CONCORDANCIA: " PALINDROMO                    :LOOP
NOTOK   SN = SN - 1                                             :(NEXT)
BAD     OUTPUT = "ENTRADA INCORRECTA: " NEWLINE                 :LOOP
END
```

Ejecución de muestra:	**Entrada:**	**Salida:**
	11011	CONCORDANCIA: 11011
	1101101	CONCORDANCIA: 11011
	11211	ENTRADA INCORRECTA: 11211

Referencia: R. E. Griswold, "A history of the SNOBOL programming language," *ACM History of Programming Languages Conference*, Los Ángeles, CA (Junio 1978) (*SIGPLAN Notices (13)8 [Agosto 1978]*), 275-308.

de María, entonces se necesita un enunciado más fuerte. Se pueden desarrollar predicados en los que participen varios hechos de la base de datos. Se desean dos padres de María que sean diferentes; por tanto, hay que escribir:

PadreDe(*X*,*María*), *PadreDe*(*Y*,*María*), *not*(*X* = *Y*).

donde la coma que separa los predicados es un operador *and* (*y*), es decir, todos los predicados deben ser ciertos para que la relación completa sea cierta.

El poder de Prolog está en la construcción de relaciones que se pueden inferir de manera lógica a partir de un conjunto conocido de hechos. Estas relaciones se pueden construir con base en otras relaciones, como en *AbueloDe*:

AbueloDe(*X*,*Y*) :– *PadreDe*(*X*,*Z*), *PadreDe*(*Z*,*Y*).

que significa que la relación *AbueloDe* se define como (es decir, se escribe como :–) dos relaciones *AbueloDe* tales que existe un objeto *Z* para el cual *Z* es el padre de *Y* y *X* es el padre de *Z*. Por tanto, *AbueloDe*(*X*,*María*) dará por resultado que *X* sea Carlos o Ana, pero *AbueloDe*(*Y*,*John*) falla, puesto que no hay un hecho así en la base de datos.

Reescritura de términos

La *reescritura de términos* es una forma restringida de concordancia de patrones que tiene numerosas aplicaciones dentro del dominio de los lenguajes de programación. Dada la cadena $a_1a_2...a_n$ y la regla se reescribe $\alpha \Rightarrow \beta$, si $\alpha = a_i$, se dice que $a_1...a_{i-1}\beta...a_n$ es una *reescritura de términos* de $a_1a_2...a_n$. La concordancia de una consulta con una regla de la base de datos es una forma de reescritura (con sustitución, véase la sección 6.3.2).

Ya se ha analizado el término reescritura en la sección 3.3.1 en el contexto de las gramáticas BNF y el análisis sintáctico. La generación de una derivación de una cadena en un lenguaje, dada cierta gramática BNF, es simplemente una forma de proceso de reescritura. Por ejemplo, dada la gramática:

$$A \rightarrow 0B \mid 1$$
$$B \rightarrow 0A$$

se puede generar la cadena 001 con la derivación siguiente:

$A \Rightarrow 0B$	usando la regla $A \rightarrow 0B$
$0B \Rightarrow 00A$	usando la regla $B \rightarrow 0A$
$00A \Rightarrow 001$	usando la regla $A \rightarrow 1$

El ML emplea el término reescritura como una forma de definición de funciones. Por ejemplo, la función factorial se puede especificar de manera bastante sencilla en ML como:

fun factorial(n:int)= **if** n=1 **then** 1 **else** n*factorial(n–1);

En este caso, el dominio para *factorial* se compone de dos conjuntos: Para $n = 1$ el valor devuelto es 1, pero para todos los demás enteros positivos, el valor devuelto es la expresión n . *factorial*$(n - 1)$. El enunciado **if** divide los dos casos. Esta función se puede ver como dos funciones separadas, una constante 1 para el conjunto $\{1\}$ y el valor $n *$ *factorial*$(n - 1)$ para los enteros restantes (es decir $\{n \mid n > 1\}$).

En vez de usar la construcción **if** para separar los subdominios, se puede emplear reescritura de términos para indicar cada caso por separado en la definición de la función:

```
fun hecho(1) =          1
  | hecho(N:int) =      N * hecho(N-1);
```

con cada subdominio separado por |. El ML reemplazará la llamada de función por la definición apropiada.

Como un ejemplo más, considérese la función *longitud* siguiente, en ML, que combina reescritura de términos con una operación más general de concordancia de patrones:

```
fun longitud(nada) =    0
| longitud(a::y) =      1+longitud(y);
```

donde *nada* indica la lista vacía y :: es el operador de concatenación de listas. Es decir, a :: $[b, c, d] = [a, b, c, d]$. En este caso, si el dominio de la función *longitud* es la lista vacía, entonces el valor es 0; de lo contrario, si el dominio es una lista con al menos una entrada (a), entonces el valor de la función es 1 más la longitud del resto de la lista. ML equipara automáticamente el argumento a *longitud* y asigna la cabeza de la lista a a y la cola de la lista a y. Esta característica se vuelve bastante importante al analizar el polimorfismo en ML (sección 8.3).

6.3.2 Unificación

Una base de datos se compone de *hechos*, como en *PadreDe*(*Ana*,*Juan*) y *reglas*, tales como:

AbueloDe(*X*,*Y*):–*PadreDe*(*X*,*Z*), *PadreDe*(*Z*,*Y*)

Una expresión que contiene una variable o más, como en *AbueloDe*(*X*,*Juan*), se conoce como una *consulta*, y representa una relación desconocida. (Se da una descripción más precisa de esto en la exposición sobre el principio de resolución, que guía la ejecución de Prolog, en la sección 9.4.4). La característica importante de Prolog es el uso de concordancia de patrones para descubrir si la consulta es resuelta por un hecho de la base de datos o si el hecho se puede deducir usando reglas de la base de datos aplicadas a otros hechos o reglas. Prolog utiliza *unificación*, o la sustitución de variables en relaciones, para concordar patrones a fin de determinar si la consulta tiene una sustitución válida congruente con las reglas y hechos de la base de datos.

Considérese la relación *PadreDe* que se dio anteriormente. Supóngase que se desea resolver la consulta:

PadreDe(*X*,*María*) = *PadreDe*(*Juan*,*Y*).

En este caso se tiene el hecho de que *PadreDe(Juan,María)* es una solución para ambas partes de la consulta.

Se dice que el *ejemplar PadreDe(Juan,María) unifica PadreDe(X,María)* y *PadreDe (Juan,Y)* con la sustitución de Juan por *X* y María por *Y*, puesto que ambas relaciones dan ese hecho al hacer la sustitución apropiada. Se puede pensar en la unificación como en una extensión de la propiedad común de sustitución.

Sustitución. La sustitución es uno de los primeros principios que se aprenden en programación. La sustitución es el principio general que se halla detrás del paso de parámetros y la expansión de macros. Por ejemplo, considérese la macro en C:[1]

```
#define mymacro(A,B,C) printf("Binary %d%d%d is %d\n",
                       A,B,C 4*(A)+2*(B)+(C))
```

Esto expresa que siempre que se usa *mimacro*(,,) en un programa en C, el primer argumento de *mimacro* reemplaza a *A* en la definición de la macro, el segundo argumento reemplaza a *B* y el tercer argumento reemplaza a *C*. Por tanto, escribir *mimacro*(1, 0, 1) es equivalente a escribir:

```
printf("Binary %d%d%d is %d\n", 1, 0, 1, 4*(1)+2*(0)+(1))
```

Puesto que la cadena de argumentos *printf* es también una sustitución, lo anterior es en realidad lo mismo que:

```
printf("Binary 101 is 5\n")
```

Los argumentos que se están sustituyendo no tienen que ser enteros simples. *Cualquier* cadena puede sustituir a *A*, *B* y *C* en la expansión de macro. Así pues, *mimacro*(*X* + *Y*,*Z*/2, *Mivar* + 3) es equivalente a:

```
printf("Binary %d%d%d is %d\n", X + Y, Z/2,Mivar + 3,
                    4 * (X + Y) + 2 * (Z/2) + (Mivar + 3))
```

El punto importante que se debe recordar es que se está sustituyendo una expresión arbitraria por una sola variable en la definición de macro.

Ampliemos ahora este problema a las dos sustituciones de macro siguientes:

```
mimacro(X + Y,Z/2,?);
mimacro(?,?,Mivar + 3);
```

Los símbolos ? representan sustituciones desconocidas. ¿Es posible que ambos enunciados de macro representen el mismo cómputo? Éste es el principio básico tras la unificación. En el ejemplo que sigue la respuesta es clara. Si *C* representa *Mivar* + 3 en la primera expansión de

[1] *printf* imprime una cadena de salida. La función imprime su primer argumento de cadena usando códigos incrustados para indicar dónde se deben poner los argumentos sucesivos en la cadena de salida. La secuencia "%d" significa imprimir el próximo argumento de la lista como un entero, \n es un carácter de final de renglón.

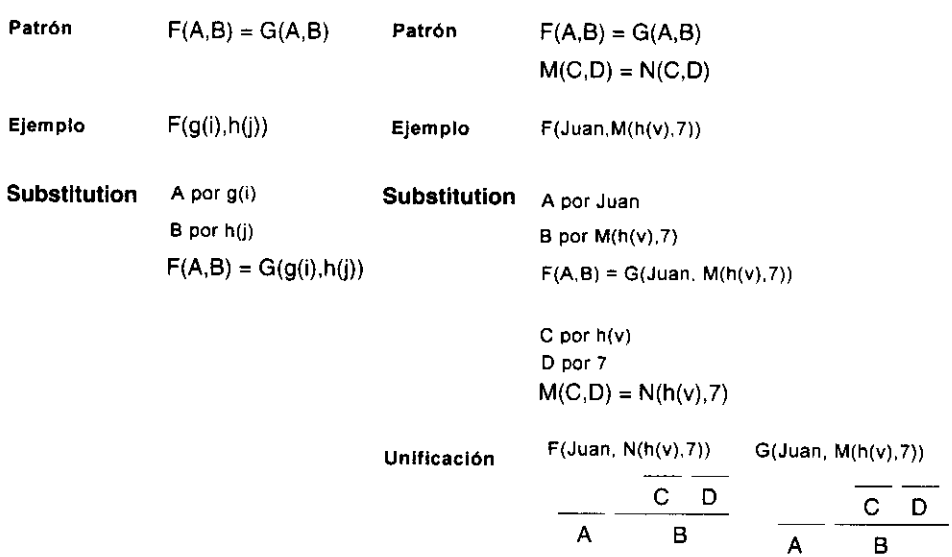


Figura 6.6. Diferencia entre sustitución y unificación.

macro, y A y B representan $X + Y$ y $Z/2$, respectivamente, en la segunda, entonces ambos enunciados de macro representan el mismo cómputo. Por otra parte, para:

$$mimacro(X + Y,Z/2,?);$$
$$mimacro(?,Z + 7,?);$$

Ningún conjunto de asignaciones a A, B o C dará por resultado que éstos representen el mismo cómputo. La determinación de un conjunto válido de sustituciones para los signos de interrogación es la esencia de la unificación. En tanto que la sustitución es el resultado de aplicar nuevos valores a argumentos de plantilla de macro, la unificación es el resultado de sustituciones simultáneas en plantillas de macro múltiples para demostrar que todas son equivalentes bajo cierto conjunto de sustituciones simultáneas.

Unificación general. Como se ha indicado antes, para unificar dos expresiones U y V, se deben encontrar sustituciones para las variables presentes en U y V que hagan idénticas las dos expresiones. Por ejemplo, para unificar $f(X, Juan)$ con $f(g(Juan), Z)$, se debe enlazar X a $g(Juan)$ y Z a *Juan* para obtener $f(g(Juan),Juan)$ como el ejemplar unificado de ambas expresiones. Si se unifican las expresiones U y V, se suele llamar s (sigma) a las sustituciones que se hacen y se escribe $U\sigma = V\sigma$.

La diferencia entre sustitución y unificación se expone en la figura 6.6. Cuando se aplica sustitución, se tiene cierta definición de patrón [por ejemplo, $F(A,B)$], que puede representar una signatura de subprograma o una definición de macro, y un ejemplar del patrón [por ejemplo, $F(g(i),h(j))$], que puede representar la invocación del subprograma o una expansión de macro. La sustitución requiere dar nuevos nombres a los parámetros de la definición de patrón

con los valores reales del ejemplar. Sin embargo, en la unificación se tiene por lo común dos definiciones de patrón individuales [por ejemplo, $F(A,B)$ y $M(C,D)$] y un ejemplar de un patrón [por ejemplo, $F(Juan,M(h(v),7))$]. Se desearía saber si existe alguna asignación a A, B, C y D que haga del ejemplar del patrón una sustitución de ambas definiciones de patrón.

Para aplicar la definición de los patrones, puede ser necesario sustituir en *ambos* sentidos. En este ejemplo, si se sustituye A por *Juan* en la definición de F, y D por 7 en la definición de M, y si también se sustituye B por los patrones $M(h(v),7)$ en la definición de F y C por $h(v)$ en la definición de M, el ejemplar de patrón representa una sustitución válida de ambas definiciones de patrón iniciales para F y M. A partir de la expresión original $F(Juan,M(h(v),7))$ se pueden obtener dos resultados distintos, los cuales dependen de si se aplica primero la definición para el patrón F o la definición para el patrón M:

Aplicar F primero: $F(Juan, M(h(v),7)) = G(Juan, M(h(v),7))$
Aplicar M primero: $F(Juan, M(h(v),7)) = F(Juan, N(h(v),7))$

Después de aplicar la segunda sustitución se obtiene el mismo resultado, $G(Juan,M(h(v),7))$. Se dice que este conjunto de sustituciones *unifica* el ejemplar de patrón tanto con F como con M.

Aplicación de la unificación a Prolog. Supóngase que se tiene la consulta q con $q = q_1, q_2$, donde q_1 y q_2 son subconsultas. Primero se intenta unificar q_1 con cierta regla p de la base de datos. Si la consulta q_1 se unifica con p en la regla:

$$p{:-}\ p_1,p_2,\ldots p_n$$

entonces se puede sustituir $q_1\sigma$ por $p\sigma$ y se obtiene la nueva consulta que se desea resolver:

$$p\sigma,\ q_2\sigma = p_1\sigma,p_2\sigma,\ldots p_n\sigma,q_2\sigma = p_1',\ p_2',\ldots p_n',\ q_2'$$

donde prima representa la consulta original modificada por las transformaciones σ.

Por otro lado, si la consulta q_1 se unifica con el hecho r, entonces se puede sustituir q_1 por *true* (*cierto*) y la consulta se vuelve ahora $true,q_2\sigma = q_2\sigma = q_2'$.

Éste es pues el proceso de ejecución para Prolog. Las consultas se unifican con reglas o hechos de la base de datos hasta que el resultado es *cierto*. Por supuesto, el resultado también puede ser *falso*, lo que significa que se unificó la regla equivocada p o el hecho erróneo r con q_1. Se debe intentar una alternativa, si existe, hasta que se encuentre una solución válida. El conjunto de transformaciones σ que se necesitan para resolver esta consulta se convierten en las "respuestas" a la consulta. El principio general que interviene aquí se llama *resolución* y se explica con más detalle en la sección 9.4.4.

Un ejemplo más completo en Prolog es el conjunto siguiente de enunciados (llamados *cláusulas* en Prolog) para definir la adición:

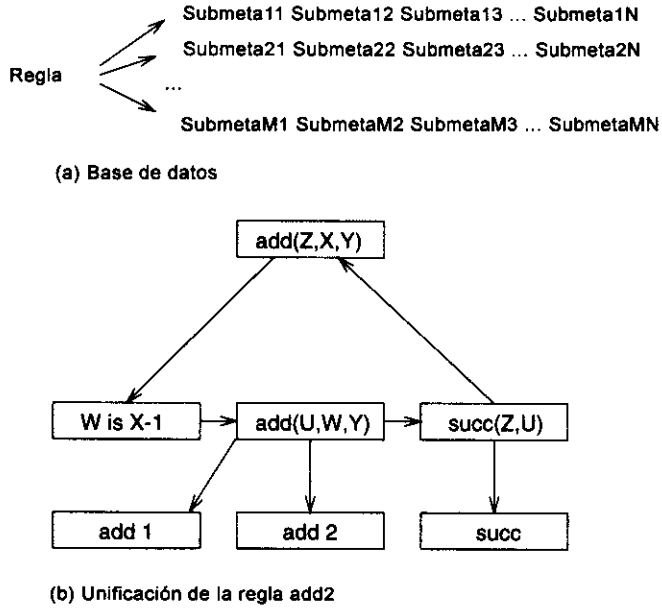


Figura 6.7. Ejecución de Prolog.

1. succ(Y,X) :–Y is X+1.
2. add(Y,0,X) :–Y=X.
3. add(Z,X,Y) :–W is X–1, add(U,W,Y), succ(Z,U).

La regla *succ* (sucesor) computa $Y = X + 1$ unificando la variable Y con la suma de X y 1. Es decir, el valor por el que se sustituye Y y que unifica Y y $X + 1$ es el valor de $X + 1$. (Recuérdese que *is* calcula el valor de la expresión, mientras que = significa la cadena sin evaluar.) Las reglas *add* calculan la suma usando dos hechos en cuanto a la adición:

$0 + x = x$ (Regla 2)
$(x + 1) + y = x + (y + 1) = x + succ(y)$ (Regla 3)

La regla 3 reduce la adición a un número igual de pasos de "restar 1 del primer término" y "sumar 1 al segundo término" hasta que el primer término es 0 (Regla 2).

Por ejemplo, para calcular la suma de 2 y 3, Y se debe unificar en la consulta *add*(Y,2,3) con las tres reglas anteriores. Prolog produce como salida:

```
add(Y,2,3).
Y = 5
```

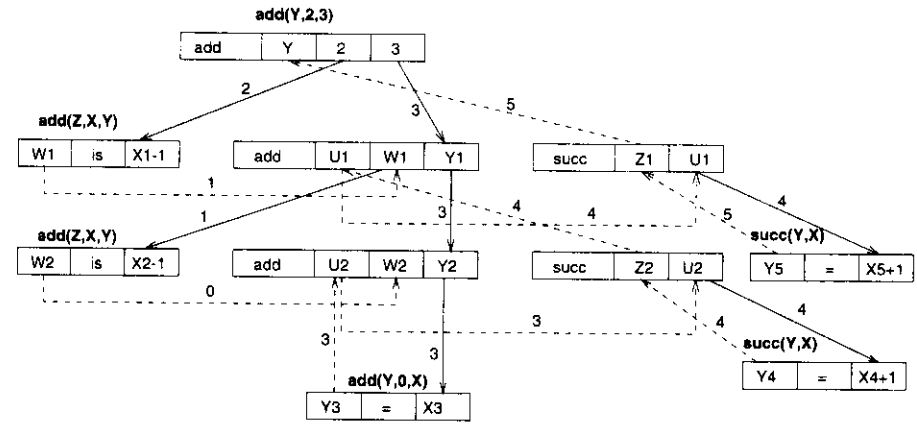


Figura 6.8. Unificación de Prolog.

donde $Y = 5$ es la transformación σ que unifica la consulta con las reglas de la base de datos que definen la adición.

Implementación. La ejecución de Prolog se sintetiza en la figura 6.7 y representa un algoritmo de recorrido de árbol normal. El conjunto de hechos en la base de datos de Prolog [figura 6.7(a)] representa un conjunto de metas *M*, una para cada regla con esa relación en la base de datos. Para el ejemplo de adición, *add* tendría dos metas posibles, *add*1 con la submeta $Y = X$, y *add*2 con las submetas W *is* $X - 1$, $add(U,W,Y)$ y $succ(Z,U)$ [figura 6.7(b)].

Para cada meta (por ejemplo, regla *add*2), Prolog trata de hacer concordar cada submeta en forma sucesiva. Primero se unifica W con $X - 1$. Esto asigna a W el valor de $X - 1$. Prolog intenta luego hacer concordar la submeta $add(U,W,Y)$ invocando las reglas *add* en forma recursiva (primero *add*1 y luego *add*2 buscando una concordancia). Si tiene éxito, U es la suma de W y Y (es decir, $(X - 1) + Y$). Si se iguala esa submeta, entonces $succ(Z,U)$ unifica Z con $U + 1$ o Z se fija en $((X - 1) + Y + 1) = (X + Y)$. Se iguala la última submeta y la regla *add*2 tiene éxito unificando Z con la suma de X y Y.

Si cualquier submeta fracasa, entonces se aplica un algoritmo normal de retroceso. Se revisa la submeta anterior en busca de una concordancia alternativa. Si no se encuentra alguna, entonces la regla misma fracasa y se ejecuta otra regla *add*, si la hay.

A través del uso de este algoritmo de recorrido de árbol general, $add(Y,2,3)$ se evalúa como sigue (figura 6.8):

1. $add(Y,2,3)$ intenta igualar la Regla 3. A W_1 se le asigna el valor $2 - 1 = 1$.
2. Prolog intenta unificar $add(U_1,1,3)$, $succ(Z_1,U_1)$.

3. $add(U_1,1,3)$ fija W_2 en 0 e intenta unificar $add(U_2,0,3)$, $succ(Z_2,U_2)$.
4. $add(U_2,0,3)$ tiene éxito unificando U_2 con 3 usando la primera regla *add*.
5. $succ(Z_2,U_2) = succ(Z_2,3)$ tiene éxito unificando Z_2 con 4.
6. Z_2 (es decir, 4) es unificado entonces con U_1 con base en la relación $add(U_1,1,3)$.
7. $succ(Z_1,U_1)$ unifica Z_1 con $U_1 + 1 = 4 + 1 = 5$.
8. Finalmente, Y es unificado con $Z_1 = 5$, que es el valor que se imprime.

Con base en este ejemplo, debe quedar claro que las pilas desempeñan un papel importante en la implementación de Prolog. Cada cláusula de Prolog es apilada, al igual que cada submeta, hasta que se encuentra una concordancia de metas.

6.3.3 Retroceso

Al describir el comportamiento de ejecución de Prolog para *add* en el ejemplo anterior, todas las submetas concordaron en el proceso del cómputo de $add(Y,2,3)$. Sin embargo, ¿qué ocurre si eso no es cierto?, ¿qué pasa si alguna submeta no se puede unificar con una regla de la base de datos? En ese caso, se dice que la regla fracasa.

Según se dieron en la figura 6.7, las metas se enumeraron en cierto orden $1..M$. Pero esto es arbitrario, y Prolog utiliza simplemente el orden en el que se introdujeron los hechos en la base de datos. En todo momento se está intentando hacer concordar una submeta. Si la concordancia ha de tener éxito, entonces una de las metas posibles tendrá éxito, sólo que no se sabe necesariamente cuál. Si se intenta primero la meta incorrecta, entonces esa meta fracasará. En este caso, simplemente se intenta otra meta posible.

Si se alcanza la última meta y ésta fracasa también, se dice que la submeta presente fracasa. Puesto que se ha apilado el conjunto de submetas que se está buscando, se retrocede a la submeta anterior que concordó y se intenta otra meta posible en busca de concordancia. A esto se le llama algoritmo general de *retroceso*, y se puede describir usando la figura 6.7(b).:

1. Para la $Submeta_{i,j}$ fíjese $k = 1$ como una nueva meta.
2. Inténtese sucesivamente hacer concordar $goal_k$ para $k = 1..M$ y devuélvase ya sea éxito o fracaso, según si alguna $meta_k$ tiene éxito o todas fracasan.
3. Si la $meta_k$ tiene éxito, entonces la $Submeta_{i,j}$ tiene éxito. Guárdese k y hágase concordar $Submeta_{i,j+1}$.
4. Si la $meta_k$ fracasa para todas las k, entonces la $Submeta_{i,j}$ fracasa. Retrocédase a la $Submeta_{i,j-1}$ e inténtese la meta siguiente posible $k + 1$ para esa submeta.
5. Si la $Submeta_{i,N}$ tiene éxito, entonces devuélvase éxito como resultado de la $meta_k$ padre que se estaba buscando.
6. Si la $Submeta_{i,j}$ fracasa para todas las j, entonces esta meta fracasa; devuélvase fracaso como resultado de la búsqueda de la $meta_k$ por el padre.

Este algoritmo intenta todas las concordancias posibles en busca de éxito, apilando y desapilando sucesivamente resultados parciales. Es un algoritmo de uso frecuente (aunque lento) que es base de muchas estrategias de búsqueda.

Prolog utiliza la función ! (corte) como una valla que siempre fracasa al retroceder. Por ejemplo, la regla:

A :-B,!,C,!D.

sólo tendrá éxito si la primera meta para B concuerda y la primera meta para C concuerda y la primera meta para D concuerda. Cualquier intento por hacer concordar la $Submeta_{i,j-1}$ en el algoritmo anterior, donde la $Submeta_{i,j-1}$ =! da como resultado el fracaso. Así pues, si la primera meta para D fracasa, A fracasa, porque no se intenta una alternativa de C como nueva meta. El uso apropiado de cortes impide retrocesos innecesarios en muchos algoritmos.

El retroceso es una técnica general de programación disponible en cualquier lenguaje que crea estructuras de árbol. Se pueden construir algoritmos de retroceso en todos los lenguajes de este libro. Pero en lenguajes como LISP, donde los árboles son consecuencias naturales del tipo de datos de lista integrado, es relativamente fácil implementar el retroceso. En Prolog, como se ha señalado, es una característica integrada.

6.4 CONTROL DE SECUENCIA ENTRE ENUNCIADOS

En esta sección se tratarán los mecanismos básicos que se usan para controlar el orden de ejecución de los enunciados dentro de un programa, y se deja para el próximo capítulo las estructuras más grandes de control de secuencia relacionadas con programas y subprogramas.

6.4.1 Enunciados básicos

Los resultados de cualquier programa están determinados por sus *enunciados básicos* que aplican operaciones a objetos de datos. Son ejemplos de estos enunciados básicos los enunciados de asignación, llamadas de subprograma, y enunciados de entrada y salida. Dentro de un enunciado básico, se pueden invocar series de operaciones usando expresiones, como se expuso en la sección anterior, pero, para nuestros fines actuales, cada enunciado básico se puede considerar como una unidad que representa un solo paso en el cómputo.

Asignaciones a objetos de datos

Los cambios al estado del cómputo por asignación de valores a objetos de datos es el mecanismo más importante que afecta el estado de un cómputo que realiza un programa. Existen varias formas de esta clase de enunciado:

Enunciado de asignación. En la sección 4.1.7 se analizó en forma breve el enunciado de asignación. El propósito primordial de una asignación es atribuir al valor *l* de un objeto de datos (es decir, a su localidad de memoria) el valor r (es decir, el valor del objeto de datos) de cierta expresión. La asignación es una operación fundamental definida para todos los tipos elementales de datos. La sintaxis para una asignación específica varía ampliamente:

A := B	Pascal, Ada
A = B	C, FORTRAN, PL/I, Prolog, ML, SNOBOL4
MOVE B TO A	COBOL
A ← B	APL
(SETQ A B)	LISP

En C, la asignación es simplemente un operador, de modo que se puede escribir: $c = b = 1$, que significa (de acuerdo con los valores de precedencia de la tabla 6.2) $(c = (b = 1))$, y que sufre el proceso de evaluación siguiente:

1. Se asigna a *b* el valor 1.
2. La expresión $(b = 1)$ devuelve el valor 1.
3. Se da a *c* el valor 1.

Es más frecuente, sin embargo, que la asignación se considere como un enunciado por separado. Puesto que la operación de asignación de Pascal no devuelve un resultado explícito, se usa sólo en el nivel de enunciados, en un enunciado de asignación explícito:

```
X := B + 2*C
Y := A + X;
```

Casi todos los lenguajes tienen un solo operador de asignación. C, sin embargo, tiene varios:

$A = B$	Asignar valor r de B a valor *l* de A, devolver valor r
A+= B (– =)	Incrementar (o disminuir) A en B ($A = A + B$ o $A = A - B$), devolver valor nuevo
++A (– –A)	Incrementar (o disminuir) A, luego devolver valor nuevo (p. ej., $A = A + 1$, devolver valor r de A)
A++ (A– –)	Devolver valor de A, luego incrementar (disminuir) A (devolver valor r, luego $A = A + 1$)

Si bien todos éstos son similares, cada uno tiene una semántica ligeramente distinta y puede tener un efecto diferente en diversas situaciones. Puesto que la operación básica de asignación consiste en asignar el valor r de una expresión al valor l de otra expresión, se puede conseguir mayor flexibilidad incluyendo operadores que afectan el valor l y el valor r de las variables. En C, por ejemplo, el operador unario * es un operador de "indirección" que hace que el valor r de una variable se comporte como si fuera un valor l, y el operador unario & es un operador de "dirección" que convierte un valor l en un valor r. Por ejemplo, en:

```
int i, *p
p = &i;
*p = 7
```

tenemos que:

1. *i* se declara como un entero;
2. *p* se declara como un apuntador a un entero;

3. *p* se fija de modo que apunte a *i*, es decir, el valor l de *i* se convierte en un valor r (&i) y este valor r se guarda como el valor r de *p*;

4. El valor r de *p* se convierte para que sea un valor l (**p*); es el valor l de *i*, de modo que el valor r de *i* se fija en 7.

El programa siguiente en C es un ejemplo más completo:

```
main()
{ int *p, *q, i, j;                    /*p y q apuntan a ints */
int **qq;                              /* qq es apuntador a apuntador a int */
i=1; j=2; printf("I=%d; J=%d;\n", i,j); /* Imprimir i y j */
p = & i;                               /* p= valor l de i */
q = & j;                               /* q= valor l de j */
*p =*q; printf("I=%d; J=%d;\n", i,j);  /* igual que i = j */
qq = & p;                              /* qq apunta a p */
**qq = 7; printf("I=%d; J=%d;\n", i,j);} /* igual que i = 7 */
```

con la salida:

```
I=1; J=2;
I=2; J=2;
I=7; J=2;
```

Enunciado de entrada. Casi todos los lenguajes de programación incluyen una forma de enunciados para leer datos del usuario en una terminal de archivos o de una línea de comunicaciones. Estos enunciados también cambian los valores de variables a través de asignaciones. Típicamente, la sintaxis es de la forma: *leer(archivo,datos)*. En C, una llamada de la función *printf* causa asignación a la "variable *buffer*" de un archivo. (El uso de archivos se analiza en forma más completa en la sección 4.3.12.)

Otras operaciones de asignación. La transmisión de parámetros (sección 7.3.1) se suele definir como una asignación del valor de argumento al parámetro formal. También se encuentran diversas formas de asignación implícita; por ejemplo, en SNOBOL4, cada referencia a la variable INPUT hace que se le asigne un nuevo valor a ella, la concordancia de metas en Prolog (resolución) ocasiona la asignación implícita a variables. Se suele poder asignar un valor inicial a una variable como parte de su declaración.

Formas de control de secuencia al nivel de enunciados

Se distinguen por lo común tres formas principales de control de secuencia al nivel de enunciados:

- *Composición.* Los enunciados se pueden disponer en una serie textual, de modo que se ejecuten en orden siempre que se ejecute la estructura mayor del programa que contiene la serie.
- *Alternancia.* Dos series de enunciados pueden formar alternativas de modo que se ejecute una u otra serie, pero no ambas, siempre que se ejecute la estructura mayor del programa que contiene la serie.

- *Iteración.* Una serie de enunciados se puede ejecutar en forma repetida, cero o más veces (cero significa que la ejecución se puede omitir del todo), siempre que se ejecuta la estructura mayor del programa que contiene la serie.

Al construir programas, uno se ocupa de armar los enunciados básicos que llevan a cabo el cómputo en los órdenes apropiados usando en forma repetida composición, alternancia e iteración para conseguir el efecto deseado. Dentro de cada una de estas categorías generales de formas de control, se suelen usar variantes apropiadas para fines particulares. Por ejemplo, en vez de una alternancia consistente en sólo dos alternativas, con frecuencia se requiere una compuesta de varias alternativas. Por lo común, el lenguaje de programación suministra diversas estructuras de control de secuencia destinadas a permitir la fácil expresión de estas formas de control.

Control explícito de secuencia

Los primeros lenguajes de programación tenían como modelo la máquina real subyacente encargada de ejecutar el programa. Puesto que las máquinas se componían de posiciones de memoria, los primeros lenguajes (por ejemplo, FORTRAN, ALGOL) las modelaban con tipos de datos simples traducibles directamente a objetos de máquina (por ejemplo, *float* de C y *real* de FORTRAN a *punto flotante* de hardware, *int* de C a *entero* de hardware) y con enunciados simples compuestos de etiquetas y bifurcaciones. La transferencia del control se indica más a menudo por el uso de un enunciado **goto** a un enunciado explícito con un nombre de etiqueta dado.

Enunciado goto. En muchos lenguajes están presentes dos formas de enunciado **goto** (**ir a**):

Goto incondicional. Dentro de una serie de enunciados, un **goto** incondicional, como:

goto PRÓXIMO

transfiere el control al enunciado con etiqueta *PRÓXIMO*. El enunciado que sigue a **goto** no se ejecuta como parte de la serie.

Goto condicional. Dentro de una serie de enunciados, un **goto** condicional, como:

if A = 0 **then goto** PRÓXIMO

transfiere el control al enunciado con etiqueta *PRÓXIMO* sólo si se cumple la condición especificada.

Aunque es una forma de enunciado simple, el **goto** la ha pasado difícil durante los últimos 25 años. Su uso en programación ha sido blanco de muchas críticas. El uso de enunciados **goto** conduce a un diseño no estructurado del programa, como se describe en seguida. Por ejemplo, gran parte del modelado formal que conduce a un modelo de corrección axiomática del diseño de programas (sección 9.4.2) depende de un estructura razonable de control para un programa. De hecho, el modelo axiomático dado en esa sección no permite incluir con facilidad un enunciado **goto**.

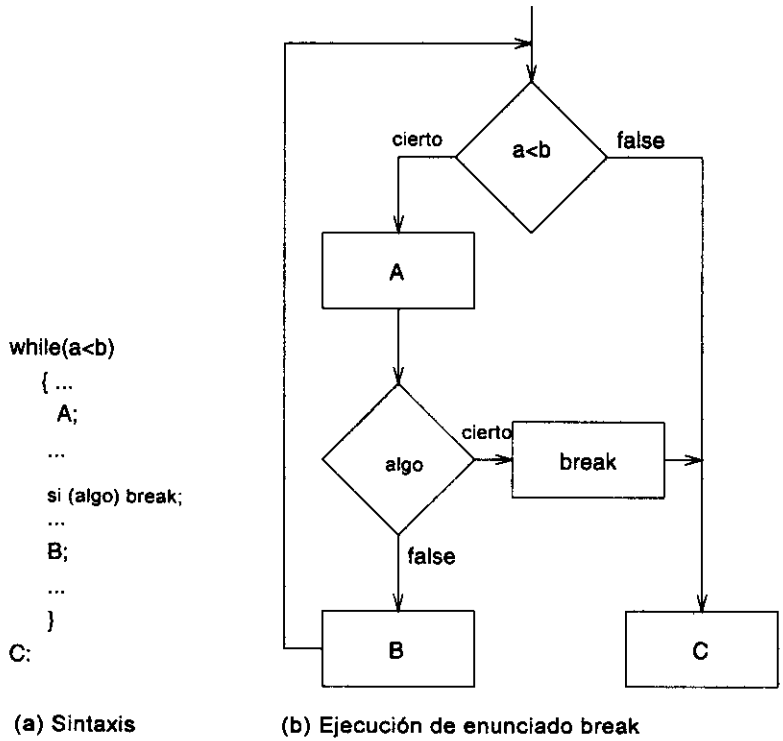


Figura 6.9. Enunciado break estructurado.

Algo más importante es que se ha demostrado que el enunciado **goto** es superfluo. Los programas se pueden escribir con la misma facilidad sin enunciados **goto** y la mayoría de los estudiantes aprenden a programar en C o Pascal sin habérseles enseñado nunca que ambos lenguajes contienen enunciados **goto**.

En tanto los lenguajes contengan *estructuras de control anidables*, como las construcciones **while** e **if**, las cuales se analizarán en breve, el **goto** es un artefacto que se puede pasar por alto sin dificultad. Desafortunadamente, en ciertos lenguajes, como las primeras versiones de FORTRAN y APL, la transferencia explícita del control es necesaria, puesto que faltan las estructuras compuestas apropiadas.

Enunciado break. Ciertos lenguajes, como C, incluyen un enunciado **break (escapar)** como una forma de control explícito estructurado. Por lo común, el **break** causa que el control pase más adelante en el programa hasta un punto explícito al final de una estructura de control dada. Este **break** en C hace que el control salga del enunciado **while, for** o **switch** que lo encierra directamente. Esto todavía proporciona una estructura de control de una salida y una entrada, que permite desarrollar las propiedades formales de un programa (figura 6.9).

El C también incluye un enunciado **continue (continuar)** relacionado. Éste hace que el control pase al inicio del cuerpo de la iteración vigente en un enunciado **while** o **for**. Así

pues, un **break** causa la salida desde una iteración, mientras que el **continue** ocasiona una nueva iteración.

Diseño de programación estructurada

Aunque casi todos los programas definen etiquetas y enunciados **goto**, la década de 1970 fue testigo de considerable controversia acerca de su uso. En ciertos lenguajes nuevos (por ejemplo, ML), los **goto** se eliminan por completo. Algunas de las ventajas aparentes de los **goto** son: (1) manejo directo por hardware para una ejecución eficiente si las etiquetas son simplemente marcas sintácticas locales en los enunciados, (2) sencillos y fáciles de usar en programas *pequeños*, (3) familiares para los programadores capacitados en lenguaje ensamblador o lenguajes más antiguos, y (4) de uso completamente general como bloque de construcción para representar (simular) cualquiera de las otras formas de control que se analizan más adelante.

Sin embargo, las desventajas del uso de enunciados **goto** sobrepasan con mucho cualquiera de estas ventajas:

Carencia de estructura jerárquica de programas. La ejecución en computadora se ha vuelto económica, en tanto que los salarios de los programadores van en aumento. Además, las computadoras se están usando en aplicaciones más críticas, como el control de aviones, maquinaria y automóviles. Es más importante que el programa se ejecute correctamente en vez de tener la máxima eficiencia. El diseño de lenguajes de programación debe reflejar estas necesidades.

El concepto de la *estructura de control con un punto de entrada y uno de salida* contribuye a un diseño más inteligible. Un programa de más de unos pocos enunciados es difícil de entender a menos que los enunciados estén organizados jerárquicamente en grupos, donde cada grupo represente una unidad conceptual del cómputo subyacente. A su vez, cada uno de estos grupos está organizado como unos cuantos subgrupos que utilizan una de las formas de control, y así sucesivamente. En el diseño de un programa, esta clase de organización jerárquica es esencial para que el programador pueda entender cómo encajan entre sí todas las partes del programa. Cuando un diseño de programa que está organizado jerárquicamente en esta forma se escribe como un programa que contiene sólo enunciados básicos y **goto**, la estructura jerárquica queda oculta en gran medida. El programador original puede ver todavía la estructura, pero, para otros, resulta difícil encontrarla. Se dice que el programa tiene una estructura muy "plana"; todos los enunciados parecen estar en un mismo nivel en lugar de hallarse organizados jerárquicamente.

No es necesario que el orden de los enunciados en el programa corresponda al orden de ejecución. Mediante el uso de enunciados **goto**, es fácil escribir programas en los cuales el control salta entre diferentes series de enunciados siguiendo patrones irregulares. En este caso, el orden en el que los enunciados aparecen en el programa tiene poca relación con el orden de ejecución de los mismos. Para entender un programa, se debe comprender el orden de ejecución de los enunciados, y es mucho más sencillo entender un programa donde los enunciados aparecen aproximadamente en el orden en el cual se ejecutan. En la figura 2.6 llamamos *código espagueti* a los programas que carecen de esta propiedad.

Los grupos de enunciados pueden servir para fines múltiples. Un programa se entiende con más facilidad si cada grupo de enunciados sirve para un solo propósito dentro de la

estructura global del programa, es decir, computa una parte individual claramente definida del cómputo completo. Suele ocurrir que dos grupos individuales de enunciados contengan varios enunciados que son idénticos en ambos. A través del uso de enunciados **goto**, se pueden combinar dos grupos de enunciados de este tipo de manera que los enunciados idénticos se escriban una sola vez y el control se transfiera a este conjunto común durante la ejecución de cada grupo. El hecho de que los enunciados tengan propósitos múltiples hace que se dificulte modificar el programa. Cuando se cambia un enunciado de un grupo de propósitos múltiples para corregir o modificar el grupo para un fin (por ejemplo, una ruta de ejecución), el uso del mismo grupo en otras secuencias de ejecución puede sufrir un deterioro sutil.

Programación estructurada. Este término se usa para el diseño de programas que hace énfasis en (1) el diseño jerárquico de estructuras de programa usando sólo las formas simples de control, de composición, alternancia e iteración ya descritas. (2) la representación directa del diseño jerárquico en el texto del programa, usando los enunciados de control "estructurados" que se describen más adelante; (3) el texto de programa en el cual el orden textual de los enunciados corresponde al orden de ejecución; y (4) el uso de grupos de enunciados de propósito único, aun cuando sea necesario copiar los enunciados. Cuando un programa se escribe siguiendo estos principios de programación estructurada, por lo común es mucho más fácil de entender, depurar, verificar que es correcto, y más adelante modificarlo y verificarlo otra vez. En la sección 6.4.3 se presenta un modelo para estructuras de control, llamado programa primo, el cual ayuda a definir formalmente lo que se quiere decir con programa estructurado.

6.4.2 Control de secuencia estructurado

Casi todos los lenguajes suministran un conjunto de enunciados de control para expresar las formas básicas de control: composición, alternancia e iteración. Un aspecto importante de los enunciados que se analizan en seguida es que cada uno es un enunciado de *entrada por salida*, lo que significa que en cada enunciado hay un solo punto de entrada al enunciado y un solo punto de salida del mismo. Si uno de estos enunciados se coloca en serie con algunos otros enunciados, entonces el orden de ejecución avanzará necesariamente desde el enunciado precedente al interior del enunciado de entrada por salida, a través del mismo y saliendo hacia el enunciado siguiente (siempre y cuando el enunciado no pueda incluir un enunciado **goto** interno que envíe el control a otra parte). Al leer un programa construido exclusivamente a partir de enunciados de entrada por salida, el flujo de la ejecución del programa deberá coincidir con el orden de los enunciados en el texto del programa. Cada enunciado de control de entrada por salida puede incluir bifurcaciones e iteraciones internas, pero el control sólo puede salir del enunciado a través de su único punto de salida.

Los lenguajes más antiguos, como COBOL y FORTRAN, contienen algunos enunciados de control de entrada por salida, pero todavía se apoyan fuertemente en enunciados **goto** y etiquetas de enunciado. Ha sido difícil adaptar ambos lenguajes a los conceptos modernos de lenguaje. El COBOL se resume en la figura 6.10 y en la sinopsis de lenguaje 6.4.

Enunciados compuestos

Un *enunciado compuesto* es una serie de enunciados que se puede tratar como un solo enunciado en la construcción de enunciados más grandes. Un enunciado compuesto se suele escribir así:

```
begin
...        – Serie de enunciados (uno o más)
end
```

En C se escribe simplemente como {...}.

Dentro del enunciado compuesto, los enunciados se escriben en el orden en que se ejecutan. Así pues, el enunciado compuesto es la estructura básica para representar la *composición* de enunciados. Puesto que un enunciado compuesto es él mismo un enunciado, los grupos de enunciados que representan unidades conceptuales de cómputo individuales se pueden conservar juntos como una unidad encerrándolos entre los corchetes **begin** ... **end**, y se pueden construir jerarquías de estos grupos.

Un enunciado compuesto se implementa en una computadora convencional colocando los bloques de código ejecutable que representan cada enunciado constitutivo en serie en la memoria. El orden en que aparecen en la memoria determina el orden en que se ejecutan.

Enunciados condicionales

Un *enunciado condicional* es uno que expresa alternancia de dos o más enunciados, o ejecución opcional de un solo enunciado; donde *enunciado* significa ya sea un enunciado básico individual, un enunciado compuesto u otro enunciado de control. La elección de la alternativa es controlada por una prueba sobre cierta condición, la cual se escribe ordinariamente como una expresión en la que intervienen operaciones relacionales y booleanas. Las formas más comunes de enunciado condicional son los enunciados **if** (si) y **case** (en caso de).

Enunciados if. La ejecución opcional de un enunciado se expresa como un **if** *de una bifurcación*:

if *condición* **then** enunciado **endif**

en tanto que una opción entre dos alternativas emplea un **if** *de dos bifurcaciones*:

if *condición* **then** $enunciado_1$ **else** $enunciado_2$ **endif**

En el primer caso, una condición cuya evaluación es 'cierta' causa la ejecución del *enunciado*, en tanto que una condición 'falsa' hace que el enunciado se pase por alto. En el **if** de dos bifurcaciones, se ejecuta el $enunciado_1$ o el $enunciado_2$, en función de si la *condición* es cierta o falsa.

Una opción entre muchas alternativas se puede expresar anidando enunciados **if** adicionales dentro de los enunciados alternativos de un solo **if**, o por medio de un **if** *de ramificaciones múltiples*:

Sinopsis de lenguaje 6.4: COBOL

Características: El COBOL (COmmon Business Oriented Language) se ha usado ampliamente desde principios de los años sesenta para aplicaciones de negocios de las computadoras.

Historia: COBOL ha evolucionado a través de una serie de revisiones de diseño, a partir de la primera versión en 1960 y revisiones posteriores en 1974 y 1984. El desarrollo del COBOL fue organizado por el Departamento de Defensa de E.U. bajo la dirección de Grace Hopper. Algunas de las ideas contenidas en COBOL se desarrollaron a partir del FLOWMATIC de Univac, incluido el uso de nombres y verbos para describir acciones y la separación de descripciones de datos con comandos —dos atributos esenciales de COBOL—. Un objetivo singular de COBOL era desarrollar un lenguaje programado en "inglés natural". Si bien el lenguaje resultante es más o menos legible, tiene una sintaxis formal y, sin capacitación adecuada, no se puede programar con facilidad.

La traducción de COBOL a código ejecutable eficiente es compleja a causa del número de diferentes representaciones de datos y la gran cantidad de opciones para la mayoría de los enunciados. Casi todos los primeros compiladores de COBOL eran extremadamente lentos, pero los desarrollos más recientes en cuanto a técnicas de compilación han conducido a compiladores de COBOL relativamente rápidos que producen código ejecutable de una eficiencia razonable.

Ejemplo: Los programas en COBOL están organizados en cuatro *divisiones*. Esta organización es resultado de dos objetivos de diseño: separar los elementos de programa *dependientes de la máquina* de los *independientes de la máquina*, y separar las descripciones de datos de las descripciones de algoritmos. El resultado es una organización tripartita del programa: la división de PROCEDURE (PROCEDIMIENTO) contiene los algoritmos, la división de DATA (DATOS) contiene descripciones de datos, y la división de ENVIRONMENT (ENTORNO) contiene especificaciones de programa dependientes de la máquina, tales como las conexiones entre el programa y los archivos externos de datos. Una cuarta división de *IDENTIFICATION* (IDENTIFICACIÓN) sirve para dar nombre al programa y su autor y para proporcionar otros comentarios como documentación del programa.

El diseño del COBOL se basa en una estructura estática en tiempo de ejecución. No se requiere gestión de almacenamiento en tiempo de ejecución, y muchos aspectos del lenguaje están planeados para permitir el uso de estructuras relativamente eficientes en tiempo de ejecución (aunque este objetivo es menos importante que el de la independencia del hardware y la transportabilidad del programa).

El lenguaje emplea una sintaxis parecida a la del idioma inglés, lo que hace que casi todos los programas sean relativamente fáciles de leer. El lenguaje suministra numerosas *palabras de sonido* optativas que se pueden usar para mejorar la legibilidad. La sintaxis hace que la escritura de los programas en COBOL sea relativamente fácil aunque tediosa, porque aun el programa más simple se hace bastante largo. La figura 6.10 ofrece una breve perspectiva de la sintaxis del COBOL.

Referencia: J. E. Sammet, "The early history of COBOL," *ACM History of Programming Languages Conference*, Los Ángeles, CA (Junio 1978) (*SIGPLAN Notices (13)8 [Agosto 1978]*), 121-161.

```
1       IDENTIFICATION DIVISION.
2       PROGRAM-ID. SUMA-DE-PRECIOS.
3       AUTOR. T-PRATT.
4       ENVIRONMENT DIVISION.
5       CONFIGURATION SECTION.
6       SOURCE-COMPUTER.  SUN.
7       OBJECT-COMPUTER.  SUN.
8       INPUT-OUTPUT SECTION.
9       FILE-CONTROL.
10           SELECT DATOS-ENT ASSIGN TO INPUT.
11           SELECT ARCHIVO-RESULTADOS ASSIGN TO OUTPUT.
12      DATA DIVISION.
13      FILE SECTION.
14      FD DATOS-ENT LABEL RECORD IS OMMITED.
15      01 PRECIO-ARTÍCULO.
16           02 ARTÍCULO PICTURE X(30).
17           02 PRECIO PICTURE 9999V99.
18      WORKING-STORAGE SECTION.
19      77 TOT PICTURE 9999V99, VALUE 0, USAGE IS COMPUTATIONAL.
20      01 RENGLÓN-DE-SUMA.
21           02 FILLER VALUE '  SUMA ='PICTURE X(12).
22           02 SALIDA-SUMA PICTURE $$,$$$,$$9.99.
23           02 SALIDA-CUENTA PICTURE ZZZ9.
24           ... Más datos
25      PROCEDURE DIVISION.
26      START.
27           OPEN INPUT DATOS-ENT AND OUTPUT ARCHIVO-RESULTADOS.
28      LEER-DATOS.
29           READ DATOS-ENT AT FND GO TO RENGLÓN-DE-IMPRESIÓN.
30           ADD PRECIO A TOT.
31           ADD 1 TO CUENTA.
32           MOVE PRECIO TO SALIDA-PRECIO.
33           MOVE ARTÍCULO A SALIDA-ARTÍCULO.
34           WRITE RENGLÓN-DE-RESULTADO FROM RENGLÓN-DE-ARTÍCULO.
35           GO TO LEER-DATOS.
36      RENGLÓN-DE-IMPRESIÓN.
37           MOVE TOT TO SALIDA-SUMA.
38           ... Más enunciados
39           CLOSE DATOS-ENT Y ARCHIVO-RESULTADOS.
40           STOP RUN.
```

Figura 6.10. Muestra de texto en COBOL.

```
if condición1 then enunciado1
    elsif condición2 then enunciado2
    ...
    elsif condiciónn then enunciadon
    else enunciadon+1 endif
```

Enunciados case. En un **if** de ramificaciones múltiples las condiciones suelen adoptar la forma de prueba repetida del valor de una variable, por ejemplo:

```
if Marca = 0 then enunciado0
    elsif Marca = 1 then enunciado1
    elsif Marca = 2 then enunciado2
    else enunciado3
    endif
```

Esta estructura común se expresa de manera más concisa como un enunciado **case**, como en Ada:

```
case Marca is
    when 0 => begin
        enunciado0
        end;
    when 1 => begin
        enunciado1
        end;
    when 2 => begin
        enunciado2
        end;
when others => begin
        enunciado3
        end;
end case
```

En general, la variable *Marca* se puede sustituir por cualquier expresión cuya evaluación sea un solo valor, y entonces las acciones para cada uno de los valores posibles se representan mediante un enunciado compuesto precedido por el valor de la expresión que causaría la ejecución de ese enunciado compuesto. Los tipos de enumeración y los subámbitos de enteros son particularmente útiles para establecer los valores posibles que la expresión puede devolver en un enunciado **case**. Por ejemplo, si la variable *Marca* antes citada se define con el subámbito 0.5 como su tipo, entonces, durante la ejecución del enunciado **case**, los valores para *Marca* de 0, 1 o 2 causarán la ejecución del $enunciado_0$, $enunciado_1$ o $enunciado_2$, respectivamente, y cualquier otro valor hará que se ejecute el $enunciado_3$.

Implementación. Los enunciados **if** se implementan con facilidad usando las instrucciones usuales de bifurcación y salto manejadas por hardware (la forma de hardware de **goto**

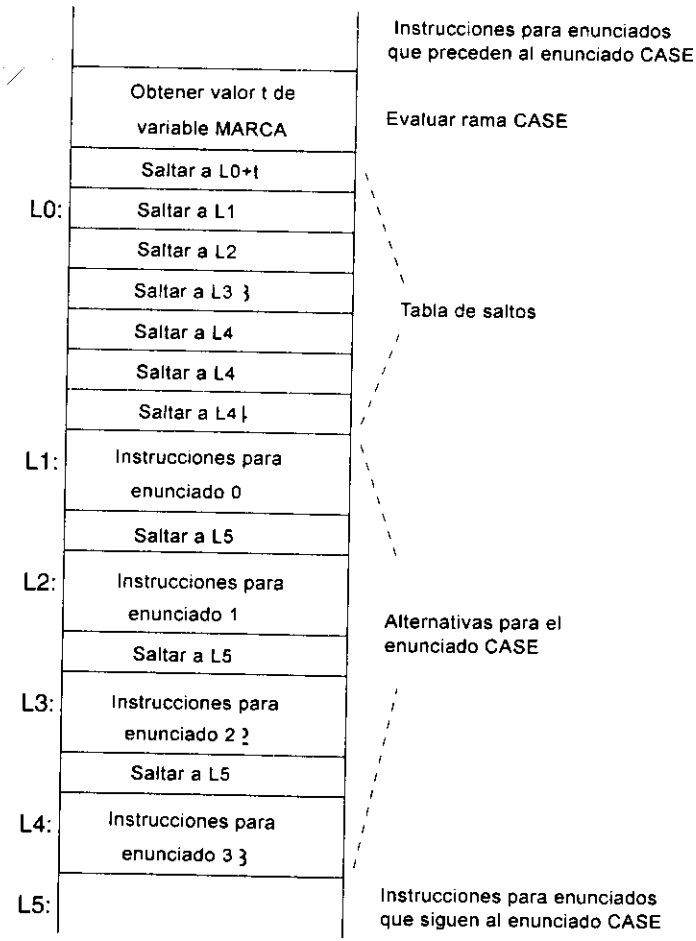


Figura 6.11. Implementación por tabla de saltos de un enunciado **case**.

condicional y no condicional). Los enunciados **case** se implementan por lo común usando una tabla de saltos para evitar la prueba repetida del valor de la misma variable. Una *tabla de saltos* es un vector, guardado en forma secuencial en la memoria, cada uno de cuyos componentes es una instrucción de salto no condicional. Se evalúa la expresión que forma la condición del enunciado **case**, y el resultado se transforma en un entero pequeño que representa el desplazamiento al interior de la tabla de saltos desde su dirección base. La instrucción de salto en ese desplazamiento, cuando se ejecuta, conduce al principio del bloque de código que representa el código por ejecutar si se elige esa alternativa. La estructura de implementación resultante para el enunciado **case** se muestra en la figura 6.11.

Enunciados de iteración

La iteración suministra el mecanismo básico para los cálculos repetidos en casi todos los programas. (Los subprogramas recursivos del próximo capítulo constituyen el otro.) La estructura básica de un enunciado de iteración consiste en una *cabeza* y un *cuerpo*. La cabeza controla el número de veces que el cuerpo se va a ejecutar, en tanto que el cuerpo es ordinariamente un enunciado (compuesto) que proporciona la acción del enunciado. Aunque los cuerpos de los enunciados de iteración están bastante libres de restricciones, por lo común sólo se usan unas pocas variantes de estructura de cabeza. Examinemos algunas que son representativas.

Repetición simple. El tipo más sencillo de cabeza de enunciado de iteración especifica que el cuerpo se debe ejecutar cierto número fijo de veces. El PERFORM de COBOL es representativo de esta construcción:

perform *cuerpo* K **times**

que hace que se evalúe *K* y luego se ejecute el cuerpo del enunciado ese número de veces.

Sin embargo, incluso este simple enunciado plantea ciertas cuestiones sutiles. ¿Se puede volver a evaluar *K* en el *cuerpo* y cambiar el número de iteraciones? ¿Qué ocurre si *K* es 0 o negativo? ¿Cómo afecta esto la ejecución?

Aunque estas preguntas pueden parecer sutilezas para este sencillo enunciado de iteración, las mismas surgen en todas las formas del enunciado, por lo cual es importante examinarlas aquí en su forma más sencilla. En cada caso es importante preguntar: (1) ¿cuándo se hace la prueba de terminación? y (2) ¿cuándo se evalúan las variables que se usan en la cabeza del enunciado?

Repetición mientras se cumple la condición. Se puede construir una iteración algo más compleja usando una cabeza de *repetir mientras*. Una forma típica es:

while *prueba* **do** *cuerpo*

En esta forma de enunciado de iteración, la expresión de prueba se reevalúa después de cada vez que se ha ejecutado el cuerpo. Adviértase también que en este caso es de esperar que la ejecución del cuerpo cambie algunos de los valores de las variables que aparecen en la expresión de prueba; de lo contrario la iteración, una vez iniciada, nunca terminaría.

Repetición mientras se incrementa un contador. La tercera forma alternativa de enunciado de iteración es el enunciado cuya cabeza especifica una variable que sirve como un contador o índice durante la iteración. En la cabeza se especifican un valor inicial, un valor final y un incremento, y el cuerpo se ejecuta repetidamente usando primero el valor inicial como valor de la variable índice, luego el valor inicial más el incremento, después el valor inicial más dos veces el incremento, y así sucesivamente, hasta que se alcanza el valor final. En FORTRAN-77 ésta es la única forma de enunciado de iteración disponible. El enunciado **for** en ALGOL ilustra la estructura típica:

for I := 1 **step** 2 **until** 30 **do** *cuerpo*

En su forma general, tanto el valor inicial como el valor final y el incremento pueden estar dados por expresiones arbitrarias, como en:

for K := N-1 **step** 2 × (W-1) **until** M× N **do** *cuerpo*

Surge una vez más la pregunta respecto a cuándo se hace la prueba de terminación y cuándo y con qué frecuencia se evalúan las diversas expresiones. En este caso, la cuestión es de importancia fundamental también para el implementador del lenguaje, porque estos enunciados de iteración son candidatos primarios a optimización, y las respuestas pueden afectar en gran medida las clases de optimizaciones que se pueden llevar a cabo.

Repetición indefinida. Cuando las condiciones para la salida de la iteración son complejas y no fácilmente expresables en la cabeza de la iteración usual, suele emplearse una iteración sin una prueba explícita de terminación en la cabeza, como en este ejemplo en Ada:

```
loop
   ...
   exit when condición;
   ...
end loop;
```

o en Pascal, usando una iteración **while** con una condición que siempre es cierta:

while cierta **do begin ... end**

El enunciado **for** de C permite todos estos conceptos en una sola construcción:

for(*expresión*$_1$;*expresión*$_2$;*expresión*$_3$){*cuerpo*}

donde *expresión*$_1$ es el valor inicial, *expresión*$_2$ es la condición de repetición y *expresión*$_3$ es el incremento. Todas estas expresiones son opcionales, lo cual permite mucha flexibilidad en las iteraciones en C. Algunas iteraciones de muestra en C se pueden especificar como:

Contador simple de 1 a 10:	**for**(i=1; i<=10; i++){*cuerpo*}
Iteración infinito:	**for**(;;){*cuerpo*}
Contador con condición de salida:	**for**(i=1;i<=100 && NoFinArchivo ;i++){*cuerpo*}

Implementación de enunciados iterativos. La implementación de enunciados de control iterativos usando la instrucción bifurcar/saltar de hardware es sencilla. Para implementar una iteración **for**, las expresiones de la cabeza de la iteración que definen el valor final y el incremento se deben evaluar a la entrada inicial de la iteración y guardarse en áreas especiales de almacenamiento temporal, de donde se pueden recuperar al inicio de cada iteración para usarse en la prueba e incremento de la variable controlada.

Problemas en el control de secuencia estructurado

Un enunciado **goto** se suele considerar como un último recurso cuando los enunciados de control estructurado antes descritos resultan inadecuados para la expresión de una estructura difícil de control de secuencia. Aunque en teoría siempre es posible expresar cualquier estructura de control de secuencia usando sólo las formas de enunciados estructurados, en la práctica una forma difícil puede no tener directamente una expresión natural si se usan sólo esos enunciados. Se conocen varias de esas áreas problema, y con frecuencia se suministran construcciones especiales de control para estos casos, las cuales hacen innecesario el uso de un enunciado **goto**. Las más comunes son:

Iteraciones de salidas múltiples. Con frecuencia, varias condiciones pueden requerir la terminación de una iteración. La iteración de búsqueda es un ejemplo común. Un vector de K elementos se va a examinar en busca del primer elemento que satisface cierta condición. La iteración termina si se alcanza el final del vector o si se encuentra el elemento apropiado. La iteración a través de los elementos de un vector se expresa de manera natural usando una iteración **for**:

```
for I := 1 to K do
   if VECT[I] = 0 then goto α {α afuera de la iteración}
```

En un lenguaje como Pascal, sin embargo, o bien se debe usar un enunciado **goto** para escapar de la parte media de la iteración, como en el caso anterior, o la iteración **for** se debe reemplazar por una iteración **while**, el cual oculta la información que contiene la cabeza de la iteración **for** acerca de la existencia y ámbito de la variable índice *I*.

El enunciado **exit** en Ada y el enunciado **break** de C proporcionan una construcción alternativa para expresar estas salidas de la iteración sin el uso de un enunciado **goto**:

```
for I in 1 .. K iteración
   exit when VECT(I) = 0;
end loop;
```

do-while-do. El lugar más natural para probar si se debe salir de una iteración no suele estar al principio o al final de la iteración, sino a la mitad, después de que se ha hecho cierto procesamiento, como en:

```
loop
   read(X)
   if fin-de-archivo then goto α {afuera de la iteración}
   process(X)
end loop;
```

A esta forma se le conoce a veces como una "do while do" (hacer mientras hacer), puesto que un **while** "intermedio" puede manejar esta secuencia:

```
dowhiledo
   read(X)
   while (no final_de_archivo)
   process(X)
end dowhiledo
```

Desafortunadamente, ningún lenguaje común implementa esta estructura, aunque, en C, **if** (*condición*) **break** se acerca y **exit when** de Ada es similar.

Condiciones excepcionales. Las excepciones pueden representar diversas condiciones de error, tales como condiciones inesperadas de final de archivo, errores de ámbito de subíndices o datos malos por procesar. Los enunciados que manejan el procesamiento de estas condiciones excepcionales se suelen agrupar en un lugar especial en el programa, como al final del subprograma en el cual se podría detectar la excepción, o posiblemente en otro subprograma que se usa sólo para manejar excepciones. La transferencia desde el punto donde se detecta la condición excepcional hasta el manejador de excepciones (grupo de enunciados) se suele representar mejor usando un enunciado **goto** en lenguajes que no implementan enunciados especiales para manejo de excepciones. Sin embargo, Ada y ML suministran mecanismos especiales de lenguaje para definir manejadores de excepciones y para especificar la transferencia de control que se necesita cuando se detecta una excepción. El enunciado **raise** de Ada es típico de este manejo de excepciones:

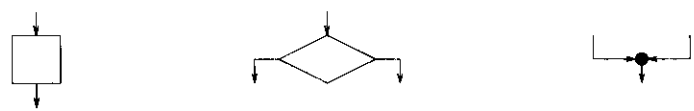
```
raise MAL_VALOR_CAR
```

Este enunciado transfiere el control a los enunciados de manejo de excepciones que están asociados con el nombre de excepción MAL_VALOR_CAR. Las excepciones y los manejadores de las mismas se analizan más a fondo en la sección 9.1.1.

6.4.3 Programas primos

Aunque la colección de estructuras de control presentadas en este capítulo parece a primera vista una recopilación aleatoria de enunciados, se puede usar la teoría de *programas primos* para describir una teoría congruente de las estructuras de control. El programa primo fue desarrollado por Maddux [MADDUX 1975] como una generalización de la programación estructurada para definir la descomposición jerárquica peculiar de un diagrama jerárquico.

Se supondrá aquí que los grafos de programas tienen tres clases de nodos:



Los nodos de función representan cómputos que hace un programa y se muestran como cuadros con un solo arco que entra a este tipo de nodo y un solo arco que sale del mismo. Intuitivamente, un nodo de función representa un enunciado de asignación, el cual causa un cambio en el estado de la máquina virtual después de su ejecución.

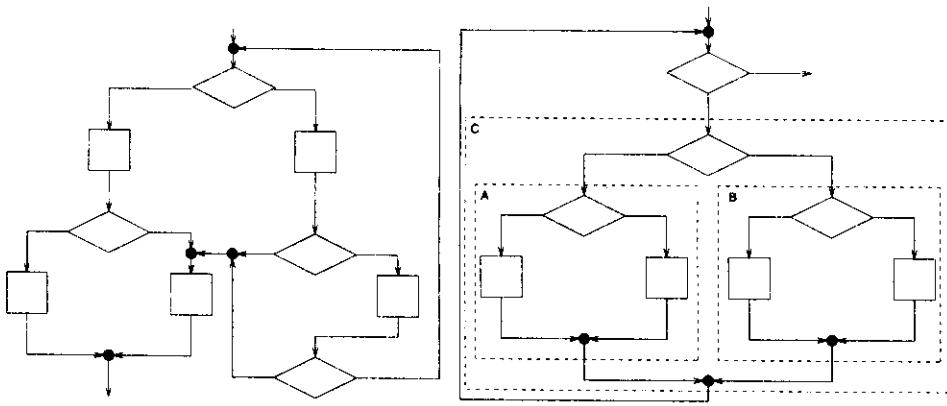


Figura 6.12. Diagramas de flujo.

Los nodos de decisión se representan como cuadros con forma de diamante con un arco de entrada y dos arcos de salida, *cierto* y *falso*. Éstos representan predicados, y el control fluye fuera de un cuadro de decisión ya sea en la rama cierta o en la falsa.

Un nodo de reunión se representa como un punto donde dos arcos fluyen juntos para formar un solo arco de salida.

Todo diagrama de flujo consiste en estos tres componentes. Definimos un *programa propio*, que es nuestro modelo formal de una estructura de control, como un diagrama de flujo que:

1. tiene un solo arco de entrada;
2. tiene un solo arco de salida; y
3. tiene una ruta del arco de entrada a cada nodo y de cada nodo al arco de salida.

El objetivo es diferenciar los programas propios "estructurados" de los "no estructurados". Por ejemplo, con referencia a la figura 6.12, hay una diferencia cualitativa obvia entre la figura 6.12(a) y la figura 6.12(b). El diagrama de flujo de la izquierda contiene una colección grande de arcos que en apariencia vagan al azar por todo el grafo, mientras que el grafo de la derecha tiene una estructura anidada ordenada. El programa primo define este concepto.

Un *programa primo* es un programa propio que no se puede subdividir en programas propios más pequeños. Si no se pueden cortar dos arcos del programa propio para separar el mismo en grafos individuales, entonces el programa propio es primo. (La excepción a esta regla es que las series largas de nodos de función se consideran como un solo primo.) La figura 6.12(a) representa un primo, en tanto que la figura 6.12(b) no es primo. Los cuadros punteados, *A*, *B* y *C*, representan todos ellos programas propios anidados dentro del diagrama de flujo global.

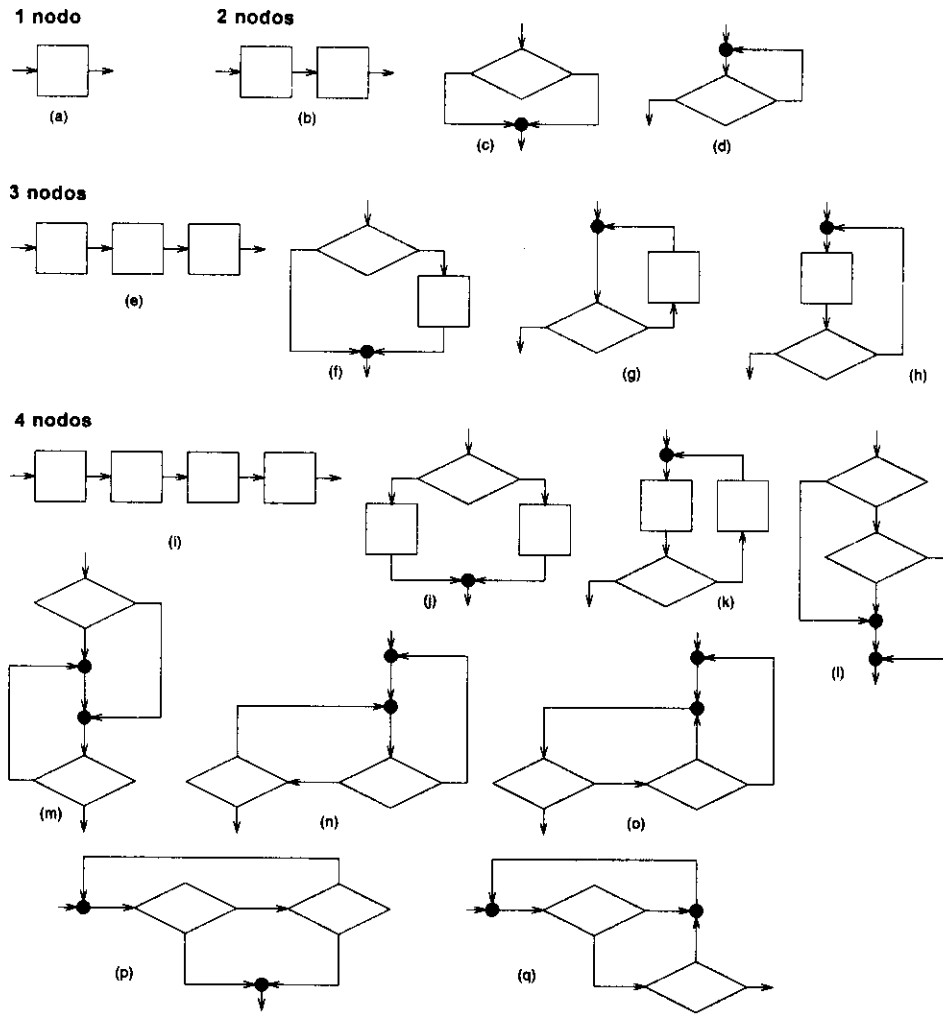


Figura 6.13. Enumeración de programas primos.

Se define un *programa compuesto* como un programa propio que no es primo. La figura 6.12(b) representa un compuesto. Al reemplazar cada componente primo de un programa propio por un nodo de función (por ejemplo, reemplazando los componentes primos *A* y *B* de la figura 6.12(b) por nodos de función) se puede repetir el proceso (por ejemplo, revelando ahora el cuadro *C* como un primo), hasta que se consigue una descomposición prima de cualquier programa propio.

Todos los primos se pueden enumerar. La figura 6.13 describe todos los programas primos de hasta 4 nodos. Obsérvese que casi todos ellos son ya sea "ineficaces" o son las estructuras

de control comunes anteriormente descritas en este capítulo. Los primos (a), (b), (e) e (i) representan todos ellos series de nodos de funciones y corresponden al bloque básico de casi todos los lenguajes. El primo (f) es el if-then, (g) es el do-while, (h) es el repeat-until, (j) es el if-then-else y (k) es el do-while-do de la sección anterior.

Considérense los primos (c), (d), y (l) a (q). Todos éstos se componen sólo de nodos de decisión y reunión. No hay un nodo de función, así que no cambian el espacio de estado de la máquina virtual. Puesto que estos primos no modifican valores de datos, todos ellos computan la función de identidad. Sin embargo, (c) y (l) siempre salen, de modo que son funciones de identidad sobre todos los datos de entrada, en tanto que los otros pueden formar iteraciones. Una vez que lo hacen, continuarán formando iteraciones y nunca saldrán. Éstos representan funciones parciales, que salen sólo para ciertos valores de datos de entrada. Ninguno de ellos representa estructuras de control eficaces en un programa.

No es sorprendente que los lenguajes de programación hayan sido definidos con las estructuras de control que se describen en este capítulo. Todas ellas representan primos con números reducidos de nodos. Estos primos son fáciles de entender, y los cambios al espacio de estado por ejecución de estos primos se vuelve manejable. Al enumerar estos primos se hace obvio que do-while-do es una estructura de control natural que por desgracia no ha sido tomada en cuenta por los planificadores de lenguajes.

El teorema de estructura. Cuando se desarrolló el concepto de programación estructurada en la década de 1970, existía preocupación acerca de si el uso de sólo estas estructuras de control limitaría el poder de los programas. Es decir, ¿se podría programar el programa de la figura 6.12(a) usando sólo las estructuras de control de la figura 6.13? Un teorema debido a Böhm y Jacobini [BOHM y JACOBINI 1966] respondió a esta pregunta. Estos autores demostraron que cualquier programa primo se podía convertir en uno que sólo utiliza enunciados **while** e **if**. La figura 6.14 es un bosquejo de un proceso similar a la construcción de Böhm-Jacobini; sin embargo, esta prueba es de Harlan Mills [LINGER *et al.* 1979]:

1. Dado cualquier diagrama de flujo, rotule cada nodo. Rotule el arco de salida con el número 0.
2. Defina *I* como una nueva variable de programa.
3. Para cada nodo del diagrama de flujo, aplique la transformación que se describe en la figura 6.14(a).
4. Reconstruya el programa como se indica en la figura 6.14(b).

Debe quedar claro que *I* opera como un contador de instrucciones de máquina virtual que indica el próximo enunciado por ejecutar, y el programa completo es simplemente una serie de enunciados **if** anidados dentro de una sola iteración **while**. Este programa funciona igual que el diagrama de flujo original con la adición de cambios a la variable *I* conforme se ejecuta.

Se ha citado el resultado de Böhm y Jacobini como una razón de por qué es *innecesario* evitar enunciados **go to**. Se puede programar cualquier algoritmo, con o sin el enunciado **go to**, y luego usar el teorema de estructura para convertir el programa en un programa "bien estructurado".

Se han vendido muchas "máquinas estructuradoras" de este tipo a los crédulos. "Programación estructurada" no es lo mismo que "buena programación". Sólo significa usar estructuras

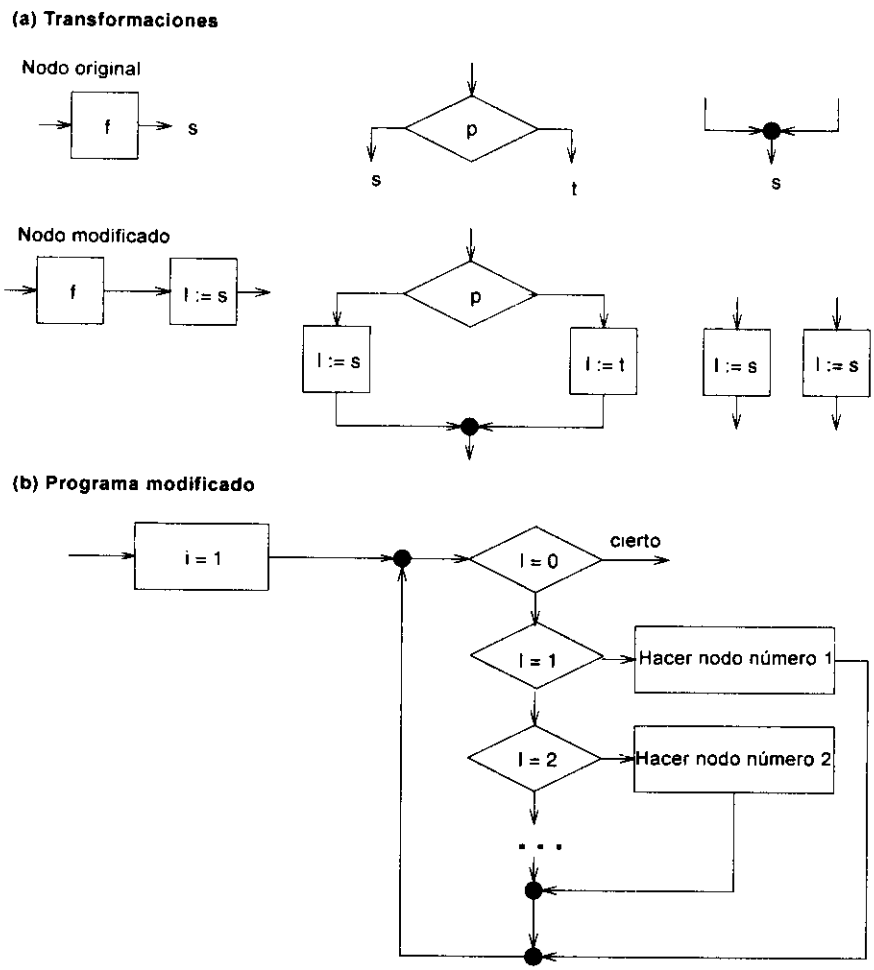


Figura 6.14. El teorema de estructura.

de control que son primos con números reducidos de nodos. Si se parte de código espagueti malo, entonces la transformación será simplemente a código estructurado malo.

Lo que proporciona el teorema de Böhm-Jacobini es una prueba de existencia de que todos los programas se *pueden* programar usando sólo las estructuras de control normales. El algoritmo no suministra la mejor solución. Es todavía al programador a quien corresponde desarrollarla. Todo lo que sabemos es que hay una solución *que no es peor que* el resultado de Böhm-Jacobini y que quizá es mucho mejor.

6.5 LECTURAS ADICIONALES SUGERIDAS

Muchos de los textos generales sobre lenguajes que se comentaron en el capítulo 1 describen cuestiones de control de secuencia. El *ACM Computing Surveys* de septiembre de 1989 [ACM 1989] presenta varios artículos sobre paradigmas de lenguajes de programación, que incluyen las estructuras de control necesarias. La relación formal entre estructuras de control y cuestiones de prueba formal de corrección se analiza en [GANNON *et al.* 1994). Las tablas de decisión son otra forma de estructura de control que se analiza en [METZNER y BARNES 1977], y [PRENNER *et al.* 1972] suministra información adicional sobre algoritmos de retroceso.

6.6 PROBLEMAS

1. En C, explique el comportamiento del fragmento de programa siguiente:

```
{int i=1
    j=2;
    if (i = j) {printf("cierto: %d %d\n",i,j);}
       else printf{falso: %d %d\n",i,j);}
```

2. Una cuarta forma de sintaxis de expresiones se llama *prefija polaca inversa*, la cual es la notación prefija usual escrita al revés. Por ejemplo, $a + (b \times c)$ es $+ a \times b\ c$ en prefija, $a\ b\ c \times +$ en posfija, y $c\ b \times a +$ en prefija polaca inversa. La prefija polaca inversa tiene propiedades similares a la notación posfija usual, pero ocasionalmente tiene ciertas ventajas sobre la posfija para generar código eficiente en lenguaje de máquina. Explique por qué.

3. Proporcione la representación de árbol para lo siguiente, suponiendo precedencia de C:

   (a) – A – B / C * D / E / F + G
   (b) A * B > C + D / E – F
   (c) ! A & B > C + D

   Repita lo anterior para la precedencia definida en Pascal y luego en Smalltalk.

4. Proporcione todos los programas primos compuestos de cinco nodos.

5. Demuestre que para cualquier $n > 1$ existe un programa primo de n nodos.

6. Uno de los resultados teóricos principales que apoyan el reemplazo de enunciados **go to** por enunciados "estructurados" se debe a un resultado obtenido por Böhm y Jacobini [BOHM y JACOBINI 1966]. Ellos probaron que cualquier programa propio también se puede programar usando sólo series de enunciados con enunciados **if** y **while**. Dibuje un

programa propio con varias iteraciones y bifurcaciones. Prográmelo usando algunos **go to** y luego usando sólo construcciones **if** y **while**. Evalúe ambos con respecto a facilidad de lectura, comprensión y escritura de estos programas.

7. El teorema de Böhm-Jacobini afirma que cualquier diagrama de flujo se puede reemplazar por uno con la misma funcionalidad que usa sólo la secuencia de enunciados **if** y **while**. Demuestre que no se necesita el **if**. Es decir, **while** es suficiente.

8. El enunciado **if** en Pascal tiene la sintaxis:

if *expresión.booleana* **then** *enunciado* **else** *enunciado*

en tanto que el **if** en Ada es:

if *expresión.booleana* **then** *enunciado* **else** *enunciado* **end if**

Comente acerca de los méritos relativos de tener un terminador explícito para el enunciado, como el **end if** de Ada.

9. Uno de los enunciados de iteración en Pascal está dado por:

for *variable.simple*: = *valor_inicial* **to** *valor_final* **do** *enunciado*

Analice los méritos y los detalles de implementación si:

(a) valor_inicial y *valor_final* se evalúan una vez, cuando se ejecuta por primera vez el enunciado **for**.
(b) valor_inicial y *valor_final* se evalúan cada vez que el programa vuelve a iterar.

10. Las reglas de Prolog se pueden ver como predicados lógicos. La regla *a*:– *b*,*c* causará que *a* tenga éxito si tanto *b* como *c* tienen éxito. Por tanto, se puede decir que *a* es cierto si la condición $b \wedge c$ es cierta.

(a) ¿En qué condiciones para *p*, *q*, *r* y *s* se satisfacen las reglas de Prolog siguientes?

(1)	*(2)*	*(3)*
x :– p,q.	x :– p,!,q.	x :– p,q.
x :– r,s.	x :– r,s.	x :– r,!,s.

(b) Repita la pregunta *(a)* si se agrega la regla *x*:–*falla* después de las dos reglas dadas. ¿Qué ocurre si se agrega esta regla primero en la base de datos?

Capítulo 7

Control de subprogramas

A lo largo de los últimos capítulos, se ha descrito el proceso a través del cual los lenguajes crean tipos de datos y objetos de datos de esos tipos. Se vio cómo los programas se codifican mediante enunciados para manipular esos datos en la memoria, y se analizaron mecanismos sencillos de activación de registros por medio de los cuales los lenguajes administran el almacenamiento para los diversos objetos de datos que se declaran. En este capítulo se examinará la interacción entre subprogramas con mayor detalle y, lo que es más importante, cómo se las arreglan para pasar datos entre unos y otros de manera estructurada y eficiente.

7.1 CONTROL DE SECUENCIA EN LOS SUBPROGRAMAS

Esta sección se ocupa de los mecanismos sencillos para establecer la secuencia entre subprogramas, es decir, de cómo un subprograma invoca a otro y el subprograma llamado regresa el control al primero. La estructura de los enunciados simples de **llamada** y **regreso** de subprogramas es común a casi todos los lenguajes de programación y es el tema de esta sección. Las estructuras de control más complejas, en las que intervienen corrutinas, excepciones, tareas y organización de llamadas de subprograma también son importantes y se tratarán en el capítulo 9.

Llamada/regreso simple de subprogramas. En programación estamos acostumbrados a ver los programas como jerarquías. Un programa se compone de un solo programa principal, el cual, durante la ejecución, puede llamar varios subprogramas, los que a su vez pueden llamar otros subsubprogramas, y así sucesivamente, a cualquier profundidad. Se espera que cada subprograma termine su ejecución en cierto punto y regrese el control al programa que lo llamó. Durante la ejecución de un subprograma, la ejecución del programa que lo llamó se detiene temporalmente. Cuando se completa la ejecución del subprograma, la ejecución del programa que lo llamó se reanuda en el punto que sigue de inmediato a la llamada del subprograma. Esta estructura de control se suele explicar por la *regla de copia*: el efecto del enunciado **call** (llamar) del subprograma es el mismo que se obtendría si el enunciado **call** se reemplazara por una copia del cuerpo del subprograma (con las sustituciones apropiadas de parámetros e identificadores en conflicto) antes de la ejecución. Vistas en esta forma, las llamadas de subprograma se pueden considerar como estructuras de control que simplemente hacen innecesario copiar grandes números de enunciados idénticos o casi idénticos que ocurren en más de un lugar en un programa.

Antes de examinar la implementación de la estructura simple de llamada-regreso que se emplea para el punto de vista de regla de copia de subprogramas, echaremos un breve vistazo a algunos de los supuestos implícitos presentes en esta perspectiva y que se pueden flexibilizar para obtener estructuras más generales de control de subprogramas:

1. *Los subprogramas no pueden ser recursivos.* Un subprograma es *directamente recursivo* si contiene una llamada a sí mismo (por ejemplo, si el subprograma B contiene el enunciado **call** B); es *indirectamente recursivo* si llama otro subprograma que llama al subprograma original o que inicia una cadena adicional de llamadas de subprograma que conduce finalmente de regreso a una llamada al subprograma original.

En el caso de llamadas simples no recursivas de subprograma, se puede aplicar la regla de copia durante la traducción para reemplazar llamadas de subprograma por copias del cuerpo del subprograma y eliminar por completo la necesidad del subprograma independiente (en principio, no en la práctica). Pero si el subprograma es directamente recursivo, entonces esto no es posible incluso en principio, porque la sustitución de llamada de subprograma por cuerpo de subprograma es obviamente interminable. Cada sustitución que elimina un enunciado **call** introduce una nueva llamada al mismo subprograma, para la cual es necesaria otra sustitución, y así sucesivamente. La recursión indirecta puede permitir la eliminación de ciertos subprogramas, pero debe conducir en último término a hacer a otros directamente recursivos. Sin embargo, muchos algoritmos son recursivos y llevan en forma natural a estructuras de subprograma recursivas.

2. *Se requieren enunciados* **call** *explícitos.* Para que la regla de copia sea aplicable, cada punto de llamada de un subprograma se debe indicar de manera explícita en el programa que se va a traducir. Pero, para un subprograma que se usa como *manejador de excepciones*, no puede haber llamadas explícitas presentes.

3. *Los subprogramas se deben ejecutar por completo a cada llamada.* En la regla de copia está implícito el supuesto de que cada subprograma se ejecuta desde el principio hasta su fin lógico cada vez que se le llama. Si se le llama una segunda vez, el subprograma inicia una nueva ejecución y se ejecuta una vez más hasta su fin lógico antes de devolver el control. Pero un subprograma que se usa como *corrutina* continúa su ejecución desde el punto de su última terminación cada vez que se le llama.

4. *Transferencia inmediata del control en el punto de llamada.* Un enunciado **call** explícito en un programa indica que el control deberá transferirse directamente al subprograma en ese punto, así que copiar el cuerpo en el programa que hace la llamada tiene el mismo efecto. Pero, para una llamada *planificada de subprograma*, la ejecución del subprograma se puede diferir hasta un momento posterior.

5. *Secuencia única de ejecución.* En cualquier punto durante la ejecución de una jerarquía de subprogramas, exactamente un programa tiene el control. La ejecución avanza en una única secuencia, desde el programa que llama al subprograma invocado y de regreso al programa que hace la llamada. Si se detiene la ejecución en algún punto, siempre se puede identificar un programa que está en ejecución (es decir, que tiene el control), un conjunto de otros cuya ejecución se ha suspendido temporalmente (el programa de llamada, el programa que llama a éste, etc.), y el resto, los cuales o nunca han sido llamados o ya se han ejecutado cabalmente. Pero los subprogramas que se usan como *tareas* se pueden ejecutar en forma simultánea, de manera que varios estén en ejecución a la vez.

De los lenguajes principales que se analizan en la parte II, sólo el FORTRAN se basa de manera directa en el punto de vista de la regla de copia de los subprogramas. Todos los demás permiten estructuras más flexibles. En el capítulo 9 se examinan las diversas estructuras de control de subprogramas que son resultado de flexibilizar cada uno de los cinco supuestos anteriores.

7.1.1 Subprogramas simples de llamada-regreso

Adviértase que aquí se hace énfasis en la estructura de control de secuencia, es decir, en los mecanismos para la transferencia del control entre programas y subprogramas. Estrechamente ligada a cada una de estas estructuras de control de secuencia está la cuestión del control de datos: transmisión de parámetros, variables globales y locales, etc. Estos temas se tratan por separado en la próxima sección, con el propósito de concentrar aquí la atención sobre los propios mecanismos de control de secuencia. Por ejemplo, incluso las llamadas simples de subprograma se presentan ordinariamente en dos formas, la *llamada de función*, para subprogramas que regresan valores directamente, y la *llamada de procedimiento* o *de subrutina* para subprogramas que operan sólo a través de efectos colaterales sobre datos compartidos. Estas distinciones se basan en métodos de control de datos; sin embargo y, en consecuencia, se tratan en esos términos en la sección siguiente. Para los fines de esta sección, los dos tipos de subprograma son idénticos en cuanto a las estructuras de control de secuencia que requieren, y por ello no se hace distinción entre los dos casos.

Implementación. Para entender la implementación de la estructura de control simple de llamada-regreso, es importante construir un modelo más completo de lo que significa decir que un programa se "está ejecutando". Para expresiones y series de enunciados, se piensa en cada una como representada por un bloque de código ejecutable en tiempo de ejecución. La ejecución de la expresión o serie de enunciados significa simplemente ejecución del código, usando un intérprete de hardware o de software, como se expuso en el capítulo 2. Para subprogramas, se necesita más:

1. Existe una distinción entre una *definición* de subprograma y una *activación* de subprograma. La definición es lo que se ve en el programa escrito, lo cual se traduce a una plantilla. Una activación se crea cada vez que se llama un subprograma, usando la plantilla creada a partir de la definición.
2. Una activación se implementa como dos partes, un *segmento de código* que contiene el código ejecutable y constantes, y un *registro de activación* que contiene datos locales, parámetros y diversos elementos de datos más.
3. El segmento de código es *invariable* durante la ejecución. Es creado por el traductor y se guarda estáticamente en la memoria. Durante la ejecución se usa pero nunca se modifica. Toda activación del subprograma utiliza el mismo segmento de código.
4. El registro de activación se *crea* de nuevo cada vez que se llama el subprograma, y se *destruye* cuando el programa regresa. En tanto el subprograma se está ejecutando, el contenido del registro de activación está cambiando constantemente conforme se hacen asignaciones a variables locales y otros objetos de datos.

Para evitar confusiones, no se puede hablar simplemente de "ejecución de un enunciado particular *S* en el subprograma", sino que se debe hablar de "ejecución de *S* durante la activación *R* del subprograma". Así pues, para seguir el rastro del punto en el que el programa "se está ejecutando", se necesitan dos datos, los cuales se consideran guardados en dos variables apuntador definidas por el sistema:

Apuntador a la instrucción presente. Los enunciados y expresiones de un subprograma se representan por medio de instrucciones de algún tipo en el código ejecutable producido por el traductor y guardado en el segmento de código. Se considera que en cualquier punto durante la ejecución existe cierta instrucción en determinado segmento de código que el intérprete de hardware o software está ejecutando actualmente (o está a punto de ejecutar). Esta instrucción se llama *instrucción presente* , y se mantiene un apuntador a ella en la variable que se conoce como *apuntador a la instrucción presente*, o CIP (en inglés, *current-instruction pointer*). El intérprete actúa obteniendo la instrucción que designa el CIP, actualizando el CIP para que apunte a la instrucción siguiente en la serie, y luego ejecutando la instrucción (la cual, a su vez, puede modificar de nuevo el CIP para efectuar un salto a alguna otra instrucción).

Apuntador del ambiente presente. Puesto que todas las activaciones del mismo subprograma utilizan el mismo segmento de código, no basta con conocer simplemente la instrucción presente que se está ejecutando; también se necesita un apuntador al registro de activación que se está usando. Por ejemplo, cuando la instrucción del código hace referencia a una variable *X*, esa variable se representa ordinariamente en el registro de activación. Cada registro de activación para ese subprograma tiene un objeto de datos distinto llamado *X*. El registro de activación representa el "ambiente de referencia" del subprograma, por lo cual un apuntador a un registro de activación se conoce comúnmente como un *apuntador del ambiente*. El apuntador al registro de activación presente (ambiente de referencia presente) se mantiene durante la ejecución en la variable que se conoce como *apuntador del ambiente presente*, o CEP (en inglés, *current-environment pointer*). El registro de activación que designa el CEP se usa para resolver la referencia a *X*.

Con los apuntadores CIP y CEP, ahora es fácil entender cómo se ejecuta un programa. Se crea un registro de activación para el programa principal (puesto que sólo hay una activación de este tipo, tal registro de activación se suele crear durante la traducción junto con el segmento de código). Se asigna un apuntador al CEP hacia él mismo. Se asigna un apuntador al CIP hacia la primera instrucción en el segmento de código para el programa principal. El intérprete se pone a trabajar, obteniendo y ejecutando instrucciones como lo designa el CIP.

Cuando se llega a una instrucción **call** de subprograma, se crea un registro de activación y un apuntador hacia él se asigna al CEP. Al CIP se asigna un apuntador hacia la primera instrucción del segmento de código para el subprograma. El intérprete continúa a partir de ese punto ejecutando instrucciones en el subprograma. Si el subprograma llama otro subprograma, se hacen nuevas asignaciones para ajustar el CIP y CEP para la activación de ese subprograma.

Para poder devolver correctamente desde una llamada de subprograma, los valores del CIP y CEP deben ser guardados en algún lugar por la instrucción **call** de subprograma antes de que se asignen los nuevos valores. Cuando se alcanza una instrucción **return** (regresar) que termina una activación de subprograma, los valores antiguos del CIP y CEP que se guardaron cuando se llamó el subprograma se deben recuperar y rehabilitar. Esta rehabilitación

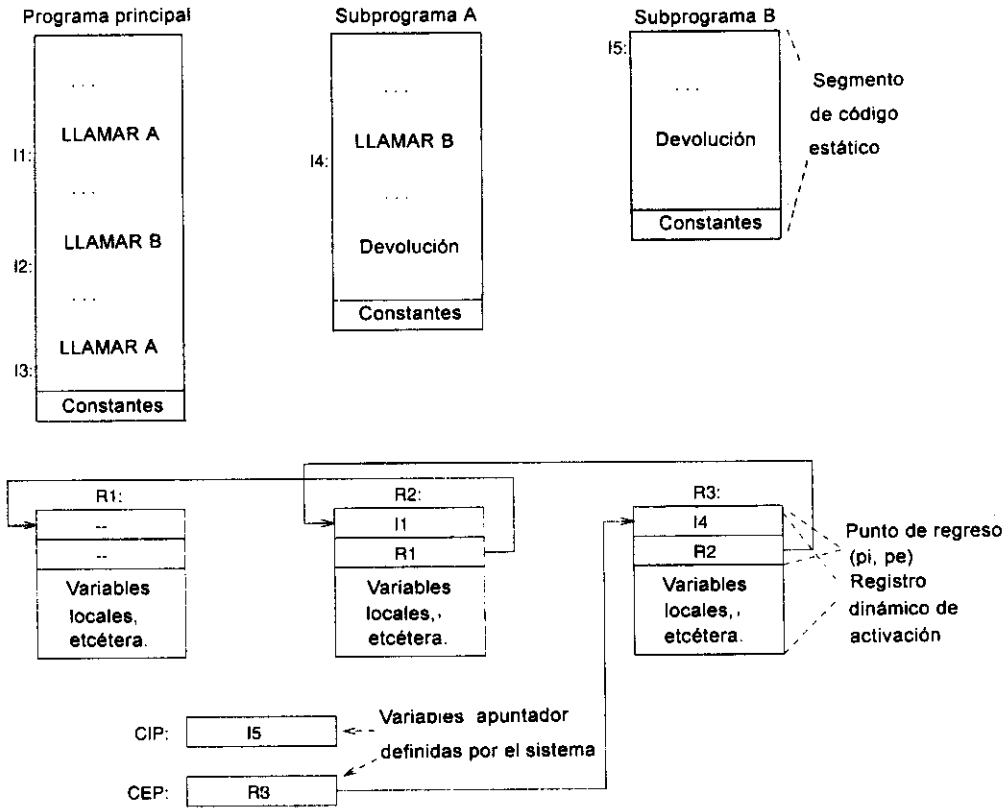


Figura 7.1. Estado de ejecución al inicio de la ejecución del subprograma B.

de los valores antiguos es todo lo que se necesita para devolver el control a la activación correcta del subprograma que llama, en el lugar adecuado, para que pueda continuar la ejecución de ese subprograma.

¿Dónde deberá guardar la instrucción **call** los valores del CIP y el CEP antes de asignar los nuevos valores? Un lugar conveniente es el registro de activación del subprograma que se está llamando. En el registro de activación se incluye un objeto de datos adicional definido por el sistema: el *punto de regreso*. El punto de regreso contiene espacio para dos valores de apuntador, la pareja apuntador de instrucción, apuntador de ambiente (*pi*, *pe*). Después que la instrucción **call** crea un registro de activación, guarda los valores antiguos (*pi*, *pe*) del CIP y CEP en el punto de regreso y asigna los nuevos (*pi*, *pe*) al CIP y CEP, con lo que efectúa la transferencia del control al subprograma llamado. La instrucción **return** obtiene los antiguos valores (*pi*, *pe*) del punto de regreso y los rehabilita como valores del CIP y el CEP, con lo que regresa el control al subprograma que llama.

Ahora bien, si observáramos el patrón global de ejecución de un programa, veríamos al intérprete ocupado, ejecutando la instrucción designada por el CIP en cada ciclo y usando el CEP para resolver referencias de datos (un tema que se trata en detalle en la próxima sección).

Las instrucciones **call** y **return** intercambian valores (*pi*, *pe*) dentro y fuera del CIP y CEP para efectuar transferencias de control entre los subprogramas. Si la ejecución se detuviera en algún punto, sería sencillo determinar cuál subprograma se estaba ejecutando en ese momento (examinando el CIP y el CEP), cuál subprograma lo había llamado (examinando el punto de regreso del subprograma en ejecución), cuál subprograma había llamado a ese subprograma (examinando su punto de regreso), y así sucesivamente. La figura 7.1 muestra esta organización para un programa principal y dos subprogramas, cada uno llamado en dos lugares.

Este modelo para la implementación de llamada y regreso de subprogramas es lo bastante general para servir como base a algunas de las variedades de estructura de control de subprogramas que se estudian más adelante. Se advierte una propiedad importante del punto de vista de regla de copia de los subprogramas: cuando mucho una activación de cualquier subprograma está en uso (es decir, se está *representando*) en cualquier punto durante la ejecución del programa. Un subprograma P puede ser llamado muchas veces distintas durante la ejecución, pero cada activación estará completa y terminada antes de que se inicie la próxima activación.

Con base en esta propiedad, se puede deducir un modelo más sencillo de implementación de subprogramas, siempre y cuando se esté dispuesto a pagar un precio en cuanto a almacenamiento con el fin de aumentar la rapidez de ejecución. La implementación más sencilla consiste en asignar almacenamiento para un solo registro de activación de cada subprograma de manera estática, como una extensión del segmento de código. En este modelo más simple (que se usa en muchas implementaciones de FORTRAN y COBOL), la ejecución del programa global se inicia con un segmento de código y un registro de activación para cada subprograma y el programa principal ya presente en la memoria. La ejecución avanza sin asigna ción dinámica de almacenamiento cuando se llama un subprograma. En su lugar, el mismo registro de activación se usa en forma repetida, simplemente inicializándolo de nuevo cada vez que se vuelve a llamar el subprograma. Como sólo una activación está en uso en cualquier punto, este uso repetido del mismo registro de activación en cada llamada no puede destruir información que se necesita de una llamada anterior, puesto que todas las llamadas anteriores ya han terminado.

Al asignar un segmento de código y un registro de activación como un bloque único de almacenamiento, se consiguen también algunas otras simplificaciones. El apuntador CEP ya no es necesario, puesto que el registro de activación actual es siempre sólo una extensión del segmento de código que designa el CIP. Una referencia en las instrucciones a una variable X siempre se puede resolver acudiendo al registro de activación anexo en vez de al CEP. Con la omisión del CEP, sólo es necesario guardar un apuntador *pi*, el CIP, y restaurarlo con la llamada y regreso del subprograma.

Con la implementación más general de llamada y regreso, el hardware subyacente suele proporcionar poco apoyo. Sin embargo, con esta implementación simplificada, el hardware suministra a menudo una *instrucción de regreso-salto* que permite implementar una llamada de subprograma en una sola instrucción de hardware. El CIP de nuestro modelo está representado directamente por el *registro de direcciones de programa* del hardware (capítulo 2). La instrucción de regreso-salto guarda el contenido de este registro de direcciones de programa en una localidad de memoria o registro interno (con frecuencia la posición de memoria inmediatamente anterior a la localidad a la cual se transfiere el control) y asigna una localidad

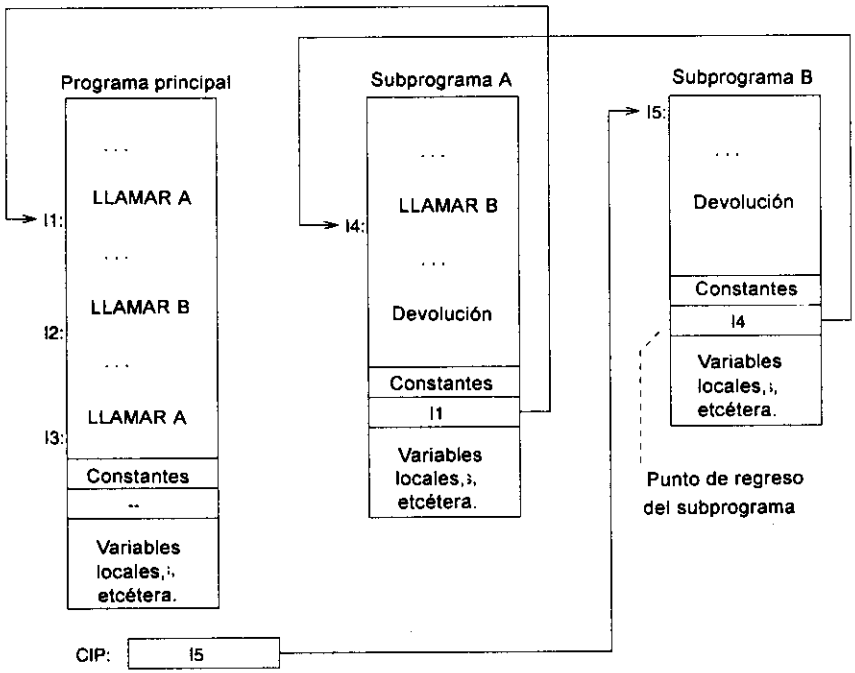


Figura 7.2. Estructura de subprograma de **call-return** (llamada-regreso).

designada como el nuevo valor del registro de direcciones de programa (con lo cual efectúa un salto a la instrucción que está en esa localidad). El efecto es exactamente el deseado: se guarda el valor antiguo del CIP, y la localidad de la primera instrucción del código de subprograma se asigna como el nuevo valor. El regreso desde un subprograma también se puede implementar ordinariamente como una sola instrucción: el valor guardado se reasigna al registro de direcciones de programa (una instrucción de salto hace esto). El resultado es una implementación simple de llamada y regreso de subprograma, a un costo en almacenamiento debido a la asignación estática de registros de activación para todos los subprogramas. En la figura 7.2 se muestra un ejemplo de esta estructura.

7.1.2 Subprogramas recursivos

Examinemos ahora uno de nuestros supuestos de subprograma, la ausencia de recursión, e investiguemos el diseño de lenguaje que permite esta característica. La recursión, en la forma de llamadas recursivas de subprograma, es una de las estructuras de control de secuencia más importantes en programación. Muchos algoritmos se representan con más naturalidad usando recursión. En LISP, donde las estructuras de lista son la principal estructura de datos disponible, la recursión es el mecanismo primario de control para series repetitivas de enunciados, y sustituye la iteración de casi todos los demás lenguajes.

Especificación. Si se permiten llamadas recursivas de subprograma, un subprograma *A* puede llamar cualquier otro subprograma, incluso *A* mismo, un subprograma *B* que llama a *A*, y así sucesivamente. Desde el punto de vista sintáctico, al escribir el programa es probable que nada cambie, pues una llamada recursiva de subprograma se ve igual que cualquier otra llamada de subprograma. En cuanto a concepto tampoco hay dificultad, siempre y cuando esté clara la distinción entre una definición y una activación de subprograma. La única diferencia entre una llamada recursiva y una llamada ordinaria es que la llamada recursiva crea una segunda activación del subprograma *durante el tiempo de vida de la primera activación*. Si la segunda activación conduce a otra llamada recursiva, entonces pueden existir tres activaciones de manera simultánea, y así sucesivamente. En general, si la ejecución del programa da por resultado una cadena de una primera llamada del subprograma *A* seguida por *k* llamadas recursivas que ocurren antes de que se haga cualquier regreso, entonces existirán $k + 1$ activaciones de *A* en el punto justo antes del regreso desde la k^a llamada recursiva. El único elemento nuevo que introduce la recursión son las múltiples activaciones del mismo subprograma, que existen todas simultáneamente en algún punto durante la ejecución.

Implementación. A causa de la posibilidad de activaciones múltiples, se necesitan los dos apuntadores, tanto CIP como CEP. Al momento de cada llamada de subprograma se crea un nuevo registro de activación, el cual se destruye posteriormente con el regreso.

Adviértase que nada de lo descrito previamente en la figura 7.1 requería que los nuevos registros de activación se crearan para los subprogramas *especiales A* y *B*. Dentro de *A*, se podría haber creado con la misma facilidad un nuevo registro de activación para *A* que uno para *B*. En el caso donde *A* llama a *B*, los *tiempos de vida no se pueden traslapar*; para cualesquiera dos activaciones de *A* y *B*, el tiempo de vida de *A incluye por completo* el de *B*. Esto simplemente expresa que si en vez de que *A* llame a *B*, *A* se llamara a sí mismo en forma recursiva, esta misma propiedad sería válida, y el nuevo registro de activación para *A* se podría agregar simplemente a la pila que contiene el registro antiguo de activación de *A*.

Cada registro de activación contiene un punto de regreso para guardar los valores de la pareja (*pi*, *pe*) usados por **call** y **return**. Si se observan sólo los valores de *pe* guardados en los puntos de regreso de la figura 7.1, se advierte que forman una lista vinculada que enlaza entre sí los registros de activación en la pila central en el orden de su creación. A partir del apuntador CEP mismo, se alcanza el registro de activación "superior" en la pila central. A partir del valor de *pe* en su punto de regreso, se puede alcanzar el segundo registro de activación en la pila; a partir del valor de *pe* en ese registro de activación se puede alcanzar el tercer registro de activación en la pila. Al final de esta cadena, el último eslabón conduce al registro de activación para el programa principal. Esta cadena de eslabones se conoce como *cadena dinámica* porque encadena entre sí activaciones en el orden de su creación dinámica durante la ejecución del programa. (En la sección 7.3.4 se analiza una *cadena estática* relacionada que vincula registros de activación entre sí para propósitos de referencia.)

El hardware convencional de computadora suministra a veces cierto apoyo de hardware para esta organización de pila central, pero en general es un poco más costosa de implementar que la estructura simple de llamada-regreso sin recursión. No existe dificultad para mezclar subprogramas implementados en la forma simple con los que utilizan una pila central, siempre

y cuando el compilador sepa cuál es cuál al compilar instrucciones **call** y **return**. Sólo los subprogramas que son llamados en efecto de manera recursiva necesitan la implementación de pila central. Así pues, en ciertos lenguajes, como PL/I, los subprograma llamados recursivamente se deben marcar con etiqueta de RECURSIVE como parte de la definición del subprograma; en otros, como C y Pascal, siempre se supone la estructura recursiva.

7.2 ATRIBUTOS DEL CONTROL DE DATOS

Las características de control de datos de un lenguaje de programación son aquellas partes que tienen que ver con la accesibilidad de datos en diferentes puntos durante la ejecución del programa. Los mecanismos de control de secuencia del capítulo anterior proporcionan los medios para coordinar el orden en el que las operaciones se invocan durante la ejecución del programa. Una vez que se alcanza una operación durante la ejecución, se le deben suministrar los datos sobre los cuales va a operar. Las características de control de datos de un lenguaje determinan cómo se pueden suministrar datos a cada operación, y cómo se puede guardar el resultado de una operación y recuperarlo para uso posterior como operando por parte de una operación subsiguiente.

Al escribir un programa, ordinariamente se tienen muy presentes las operaciones que el programa debe ejecutar, así como su orden, pero rara vez ocurre lo mismo con los operandos para esas operaciones. Por ejemplo, un programa en Pascal contiene:

$$X := Y + 2 * Z$$

Una simple inspección indica tres operaciones en serie: una multiplicación, una adición y una asignación. Pero, ¿qué hay de los operandos para estas operaciones? Es claro que un operando de la multiplicación es 2, pero los otros operandos están marcados sólo con los identificadores *X*, *Y* y *Z*, y es obvio que éstos no son los operandos, sino que sólo designan los operandos de cierta manera. *Y* podría designar un número real, un entero, o el nombre de un subprograma sin parámetros que debe ejecutarse para el cómputo de un operando. O tal vez el programador se ha equivocado y *Y* designa un valor booleano o una cadena, o sirve como etiqueta de enunciado. *Y* puede designar un valor computado cercano, quizá en el enunciado anterior, pero es igualmente probable que pueda designar un valor computado en algún punto mucho más anterior del cómputo, separado por muchos niveles de llamada de subprograma de la asignación donde se usa. Para empeorar las cosas, *Y* se puede usar como nombre de diferentes maneras en distintas secciones del programa. ¿Cuál uso de *Y* está vigente en este caso?

En pocas palabras, el problema medular del control de datos es el de saber cuál es el significado de *Y* en cada ejecución de un enunciado de asignación de esta clase. Puesto que *Y* puede ser una variable local o no local, el problema implica lo que se conoce como "papeles de alcance" para declaraciones; dado que *Y* puede ser un parámetro formal, el problema involucra técnicas para transmisión de parámetros; y puesto que *Y* puede nombrar un subprograma sin parámetros, el problema implica mecanismos para devolver resultados desde subprogramas. En la subsección siguiente se analiza en primer término lo que estos problemas son. En la sección 7.3 se describen con mayor detalle métodos para implementar estas características.

7.2.1 Nombres y ambientes de referencia

Existen básicamente sólo dos maneras de hacer que un objeto de datos esté disponible como operando para una operación.

Transmisión directa. Un objeto de datos computado en un punto como resultado de una operación se puede transmitir directamente a otra operación como operando, así, por ejemplo, el resultado de la multiplicación $2 \times Z$ se transmite directamente a la operación de adición como operando en el enunciado $X := Y + 2 \times Z$. En este caso, se asigna almacenamiento temporalmente al objeto de datos durante su tiempo de vida y puede ser que nunca se le dé un nombre.

Referencia a través de un objeto de datos con nombre. A un objeto de datos se le puede dar nombre cuando se crea, y el nombre se puede usar entonces para designarlo como un operando de una operación. Alternativamente, se puede hacer que el objeto de datos sea un componente de otro objeto de datos que tiene nombre, de modo que el nombre del objeto de datos más grande se pueda usar junto con una operación de selección para designar el objeto de datos como un operando.

La transmisión directa se usa para control de datos dentro de expresiones, pero casi todo el control de datos fuera de las expresiones implica el uso de nombres y la referencia de nombres. El problema del significado de los nombres constituye la preocupación fundamental en el control de datos.

Elementos de programa que pueden tener nombre

¿Qué clases de nombres se ven en los programas? Cada lenguaje difiere, pero algunas categorías generales que se observan en muchos lenguajes con:

1. Nombres de variables.
2. Nombres de parámetros formales.
3. Nombres de subprogramas.
4. Nombres para tipos definidos.
5. Nombres para constantes definidas.
6. Etiquetas de enunciados (nombres para enunciados).
7. Nombres de excepciones.
8. Nombres para operaciones primitivas, p. ej., +, *, SQRT.
9. Nombres para constantes de literales, p. ej., 17, 3.25.

Las categorías de la 1 a la 3, nombres para variables, parámetros formales y subprogramas, constituyen nuestra preocupación fundamental aquí. De las categorías restantes, casi todas las referencias a nombres en estos grupos se resuelven durante la traducción más bien que durante la ejecución del programa, como ya se ha expuesto. Existen ciertos casos especiales (por ejemplo, las excepciones se pueden propagar en tiempo de ejecución hacia arriba de la cadena dinámica de programas de llamada en Ada, los enunciados **goto** pueden hacer referencia a etiquetas de enunciado en otros subprogramas en Pascal, y los símbolos de operaciones

primitivas se pueden redefinir en SNOBOL4), pero estos casos especiales son relativamente raros y agregan pocos conceptos nuevos. Una vez que se entienden las ideas básicas subyacentes al control de datos para variables, parámetros formales y nombres de subprograma, la extensión a estos casos nuevos es relativamente sencilla.

Un nombre de cualquiera de las categorías anteriores se puede describir como un *nombre simple*. Un *nombre compuesto* es uno que corresponde a una estructura de datos, escrito como un nombre simple que designa la estructura de datos completa, seguido por una serie de una o más operaciones de selección que seleccionan el componente particular de la estructura nombrada. Por ejemplo, si *A* es el nombre de un arreglo, entonces *A* es un nombre simple, y *A*[3] es un nombre compuesto. Los nombres compuestos pueden ser bastante complejos, por ejemplo *A*[3].*Clase*[2].*Salón*. Las operaciones de selección y el acceso a componentes de estructuras de datos dada la estructura misma se analizan en el capítulo 4. En este capítulo, sólo queda por comentar el significado de los nombres simples. En casi todos los lenguajes, los nombres simples se representan por medio de identificadores tales como *X*, *Z2* y *Sub*1, y por tanto los términos *identificador* y *nombre simple* se usan aquí de manera indistinta.

Asociaciones y ambientes de referencia

El control de datos se ocupa en gran parte del enlace de identificadores (nombres simples) a objetos de datos y subprogramas particulares. Esta clase de enlace se conoce como una *asociación* y se puede representar como una pareja compuesta del identificador y su objeto de datos o subprograma asociado.

En el curso de ejecución de un programa en casi cualquier lenguaje, se observa que:

1. Al inicio de la ejecución del programa principal, las asociaciones de identificadores ligan cada nombre de variable declarado en el programa principal a un objeto de datos particular y unen cada nombre de subprograma invocado en el programa principal a una definición de subprograma particular.

2. Conforme el programa principal se ejecuta, éste invoca *operaciones de referencia* para determinar el objeto de datos o subprograma particular asociado con un identificador. Por ejemplo, para ejecutar la asignación:

   ```
   A := B + FN(C)
   ```

   se requieren cuatro operaciones de referencia para recuperar los objetos de datos asociados con los nombres *A*, *B* y *C* y el subprograma asociado con el nombre *FN*.

3. Cuando se llama cada subprograma, se crea un conjunto nuevo de asociaciones para ese subprograma. Cada uno de los nombres de variable y nombres de parámetro formal declarados en el subprograma está asociado con un objeto de datos particular. También se pueden crear nuevas asociaciones para nombres de subprograma.

4. Conforme el subprograma se ejecuta, él mismo invoca operaciones de referencia para determinar el objeto de datos o subprograma particular asociado con cada identificador. Algunas de las referencias pueden ser a asociaciones creadas al entrar al subprograma, en tanto que otras pueden ser a asociaciones creadas en el programa principal.

5. Cuando el subprograma regresa el control al programa principal, sus asociaciones se destruyen (o se vuelven inactivas).

6. Cuando el control regresa al programa principal, la ejecución continúa como antes, usando las asociaciones originalmente establecidas al inicio de la ejecución.

En este patrón de creación, uso y destrucción de asociaciones, se aprecian los conceptos principales del control de datos.

Ambientes de referencia. Cada programa o subprograma tiene un conjunto de asociaciones de identificador disponibles para su uso al hacer referencias durante su ejecución. Este conjunto de asociaciones de identificador se conoce como el *ambiente de referencia* del subprograma (o programa). El ambiente de referencia de un subprograma es ordinariamente invariable durante su ejecución. Se establece cuando se crea la activación del subprograma, y permanece sin cambio durante el tiempo de vida de la activación. Los valores contenidos en los diversos objetos de datos pueden cambiar, pero no así las asociaciones de nombres con objetos de datos y subprogramas. El ambiente de referencia de un subprograma puede tener varios componentes:

1. *Ambiente local de referencia* (o simplemente *ambiente local*). El conjunto de asociaciones creadas al entrar a un subprograma y que representan parámetros formales, variables locales y subprogramas definidos sólo dentro de ese subprograma conforma el *ambiente local de referencia* de esa activación del subprograma. El significado de una referencia a un nombre en el ambiente local se puede determinar sin salir de la activación del subprograma.

2. *Ambiente no local de referencia.* El conjunto de asociaciones para identificadores que se pueden usar dentro de un subprograma pero que no se crean al entrar a él se conoce como el *ambiente no local de referencia* del subprograma.

3. *Ambiente global de referencia.* Si las asociaciones creadas al inicio de la ejecución del programa principal están disponibles para usarse en un subprograma, entonces estas asociaciones forman el *ambiente global de referencia* de ese subprograma. El ambiente global es parte del ambiente no local.

4. *Ambiente predefinido de referencia.* Ciertos identificadores tienen una asociación predefinida, la cual se define directamente en la definición del lenguaje. Cualquier programa o subprograma puede usar estas asociaciones sin crearlas en forma explícita.

Visibilidad. Se dice que una asociación para un identificador es *visible* dentro de un subprograma si es parte del ambiente de referencia para ese subprograma. Una asociación que existe pero no forma parte del ambiente de referencia del subprograma actualmente en ejecución se dice que está *oculta* de ese subprograma. Con frecuencia una asociación está oculta cuando se entra a un subprograma que redefine un identificador que ya está en uso en otra parte del programa.

Alcance dinámico. Cada asociación tiene un *alcance dinámico*, el cual es la parte de la ejecución del programa durante la que esa asociación existe como parte de un ambiente de referencia. Así, el alcance dinámico de una asociación se compone del conjunto de activaciones de subprograma dentro de las cuales es visible.

Operaciones de referencia. Una *operación de referencia* es una operación con la signatura:

$$ref\_op : id \times ambiente\_de\_referencia \rightarrow objeto\ de\ datos\ o\ subprograma$$

donde *ref_op*, dado un identificador y un ambiente de referencia, encuentra la asociación apropiada para ese identificador en el ambiente y regresa el objeto de datos o la definición de subprograma asociada.

Referencias locales, no locales y globales. Una referencia a un identificador es una *referencia local* si la operación de referencia encuentra la asociación en el ambiente local; es una referencia *no local* o *global* si la asociación se encuentra en el ambiente *no local* o *global*, respectivamente. (Los términos *no local* y *global* se suelen usar de manera indistinta para indicar cualquier referencia que no es local.)

Seudónimos para objetos de datos

Durante su tiempo de vida un objeto de datos puede tener más de un nombre, por ejemplo, puede haber varias asociaciones en diferentes ambientes de referencia, donde cada uno proporciona un nombre distinto para el objeto de datos. Por ejemplo, cuando un objeto de datos se transmite "por referencia" (véase la sección 7.3.1) a un subprograma como parámetro, se puede hacer referencia a él a través de un nombre de parámetro formal en el subprograma, y también conserva su nombre original en el programa que llama. Alternativamente, un objeto de datos puede convertirse en componente de varios objetos de datos a través de vínculos de apuntador, y tener así varios nombres compuestos a través de los cuales se puede tener acceso a él. Los nombres múltiples para el mismo objeto de datos son posibles de diversas maneras en casi todos los lenguajes de programación.

Cuando un objeto de datos es visible a través de más de un nombre (simple o compuesto) *en un solo ambiente de referencia*, cada uno de los nombres se conoce como un *seudónimo* del objeto de datos. Cuando un objeto de datos tiene múltiples nombres, pero un único nombre en cada ambiente de referencia en el cual aparece, no surgen problemas. Sin embargo, la capacidad para hacer referencia al mismo objeto de datos usando nombres distintos dentro del mismo ambiente de referencia plantea serios problemas tanto para el usuario como para el implementador de un lenguaje. La figura 7.4 muestra dos programas en Pascal en los cuales

EJEMPLO 7.1. Variables de referencia en Pascal.

La figura 7.3 muestra un subprograma sencillo en Pascal con el ambiente de referencia para cada subprograma señalado. Adviértase que los identificadores A, D y C se declaran cada uno en dos lugares. El identificador A es un nombre de parámetro formal en SUB1 y también se declara como una variable en el programa principal. El identificador C es un parámetro formal en SUB2 y también un nombre de variable en el programa principal, y D es un nombre de variable local tanto en SUB1 como en SUB2. Sin embargo, en cada ambiente de referencia, sólo es visible una asociación para cada uno de estos nombres. Así, en SUB2, la asociación local para C es visible y la asociación global para C en el programa principal está oculta. En el enunciado $C := C + B$ en SUB2, aparecen tanto una referencia local a C como una referencia global a B en el programa principal.

El ambiente de referencia predefinido no se muestra. En Pascal se compone de constantes como MAXINT (el valor entero máximo) y subprogramas como *read*, *write* y *sqrt*. A cualquiera de estos identificadores predefinidos se le puede dar una nueva asociación a través de una declaración explícita de programa y, por tanto, la asociación predefinida puede quedar oculta para una parte del programa.

una variable entera tiene dos nombres *I* y *J* en diferentes puntos durante la ejecución del programa. En el primer programa no hay seudónimos, porque en ningún punto durante la ejecución se pueden usar ambos nombres *I* y *J* en el mismo subprograma. En el segundo programa, dentro del subprograma *Sub*1, *I* y *J* son seudónimos del mismo objeto de datos porque *I* es traspasado "por referencia" a *Sub*1, donde queda asociado con el nombre *J*, y al mismo tiempo *I* también es visible en *Sub*1 como nombre no local.

Los seudónimos son problemáticos para el programador porque dificultan la comprensión de un programa. Por ejemplo, si dentro de un programa uno encuentra la serie de enunciados:

```
X := A + B;
Y := C + D;
```

las asignaciones a *X* y *Y* son en apariencia independientes y podrían tener lugar en cualquier orden, o si no se hiciera referencia a la variable *X* más tarde, la primera asignación se podría eliminar del todo. Sin embargo, supóngase que *X* y *C* son seudónimos del mismo objeto de datos. Entonces los enunciados son de hecho interdependientes, y no es posible un reordenamiento o eliminación alguna sin introducir un error difícil de detectar. La posibilidad de creación de seudónimos dificulta verificar que un programa es correcto porque no se puede suponer que un par cualquiera de nombres de variable se refiere necesariamente a objetos de datos distintos.

Los problemas que los seudónimos causan al implementador son similares. Como parte de la optimización del código de programa durante la traducción, suele ser deseable reordenar los pasos de un cómputo o eliminar pasos innecesarios. Cuando es posible la creación de seudónimos, esto no se puede hacer sin un análisis adicional para asegurar que dos pasos de

```
program principal;
var A, B, C: real;
procedure Sub 1(A: real);
   var D: real:
   procedure Sub 2(C: real);
      var D: real:
      begin
      – Enunciados
      C := C+B;
      – Enunciados
      end;
   begin
   – Enunciados
   Sub2(B);
   – Enunciados
   end;
begin
– Enunciados
Sub1 (A);
– Enunciados
end
```

Ambiente de referencia para Sub2
Local C, D
No local A, Sub2 en Sub1
B, Sub1 en principal

Ambiente de referencia para Sub1
Local A, D, Sub2
No local: B, C, Sub1 en principal

Ambiente de referencia para principal
Local A, B, C, Sub1

Figura 7.3. Ambientes de referencia en un programa en Pascal.

cómputo aparentemente independientes no son dependientes a causa de la creación de seudónimos, como en el caso anterior. A causa de los problemas que ocasionan los seudónimos, los nuevos diseños de lenguajes intentan a veces restringir o eliminar del todo las características que permiten construir seudónimos.

7.2.2 Alcance estático y dinámico

El *alcance estático* de una asociación para un identificador, según se definió en la sección anterior, es el conjunto de activaciones de subprograma en las cuales la asociación es visible durante la ejecución. El alcance dinámico de una asociación siempre incluye la activación de subprograma, en la cual esa asociación se crea como parte del ambiente local. También puede ser visible como una asociación no local en otras activaciones de subprograma.

Una *regla de alcance dinámico* define el alcance dinámico de cada asociación en términos del curso dinámico de la ejecución del programa. Por ejemplo, una regla de alcance dinámico típica afirma que el alcance de una asociación creada durante una activación del subprograma P incluye no sólo esa activación, sino también cualquier activación de un subprograma llamado por P o por un subprograma llamado por P, y así sucesivamente, a menos que esa última activación de subprograma defina una nueva asociación local para el identificador que oculte la asociación original. Con esta regla, el alcance dinámico de una asociación está ligado a la *cadena dinámica* de activaciones de subprograma descrita en la sección 7.1.2.

Cuando se examina la forma escrita de un programa, el *texto del programa*, se advierte que la asociación entre referencias a identificadores y declaraciones particulares o definiciones

```
program principal(salida);
procedure Sub1(var J: integer);
   begin
   .../* J es visible, I no */
   end
procedure Sub2;
   var I: integer;
   begin
   ...
   Sub1(I); /* J es visible, I no */
   ...
   end;
begin
...
Sub2 /* Ninguna es visible */
...
end.
```

(a) Sin seudónimos

```
program principal(salida);
var I: integer;
procedure Sub1(var J: integer);
   begin
   .../* I y J se refieren a lo mismo */
   end, /* objeto de datos aquí */
procedure Sub2;
   var I: integer;
   begin
   ...
   Sub1(I); /* I es visible, J no */
   ...
   end;
begin
...
Sub2; /* I es visible, J no */
...
end.
```

(b) I y J son seudónimos en Sub1

Figura 7.4. Creación de seudónimos en un programa en Pascal.

del significado de esos identificadores también es un problema. Por ejemplo, en la figura 7.3, las referencias a B y C en el enunciado $C := C + B$ en $SUB2$ necesitan estar ligadas a declaraciones particulares de C y B como variables o parámetros formales. Pero, ¿cuáles declaraciones? Cada declaración u otra definición de un identificador dentro del texto del programa tiene un cierto alcance, llamado *alcance estático*.

Para simplificar, el término *declaración* se usa aquí para referirse a una declaración, definición de subprograma, definición de tipo, definición de constante u otro medio de definir un significado para un identificador particular dentro de un texto de programa. Una declaración crea una asociación en el texto del programa entre un identificador y cierta información acerca del objeto de datos o subprograma que va a ser nombrado por ese identificador durante la ejecución del programa. El *alcance estático* de una declaración es la parte del texto del programa donde un uso del identificador es una referencia a esa declaración particular del identificador. Una *regla de alcance estático* es una regla para determinar el alcance estático de una declaración. En Pascal, por ejemplo, se usa una regla de alcance estático para especificar que una referencia a una variable X en un subprograma P se refiere a la declaración de X al inicio del subprograma Q cuya declaración contiene la declaración de P, y así sucesivamente.

Las reglas de alcance estático relacionan referencias con declaraciones de nombres en el texto del programa; las reglas de alcance dinámico relacionan referencias con asociaciones para nombres durante la ejecución del programa. ¿Cuál debe ser la relación entre las dos? Es claro que las reglas deben ser *congruentes*. Por ejemplo, si las reglas de alcance estático para el Pascal relacionan la referencia a la variable B en el enunciado $C := C + B$ de la figura 7.3 con la declaración de B en el programa principal, entonces las reglas de alcance dinámico

también deben relacionar la referencia a *B* en tiempo de ejecución con el objeto de datos llamado *B* en el programa principal. Puede haber varias declaraciones para *B* en el texto del programa, y varios objetos de datos llamados *B* en diversas activaciones de subprograma durante la ejecución. Así pues, el mantenimiento de la congruencia entre reglas de alcance estático y dinámico no es del todo sencillo. En seguida se examinan varios enfoques distintos.

La importancia del alcance estático. Supóngase que un lenguaje no usa las reglas de alcance estático. Considérese un enunciado como $X := X + Máx$ que se presenta en un subprograma. Sin reglas de alcance estático, nada se puede determinar respecto a los nombres *X* y *Máx* durante la traducción del programa. En el transcurso de su ejecución, cuando se alcanza el enunciado, una operación de referencia debe encontrar primero las asociaciones pertinentes para *X* y *Máx*, y luego se debe determinar el tipo y otros atributos de *X* y *Máx*. ¿Existe una asociación para cada identificador? ¿Es *Máx* un nombre de subprograma, nombre de variable, etiqueta de enunciado, nombre de tipo o nombre de parámetro formal? Si *X* es un nombre de variable, ¿es de un tipo que se puede sumar a *Máx*? Ninguna de estas preguntas se puede responder hasta que se intente hacer referencia a los nombres *X* y *Máx* durante la ejecución. Más aún, cada vez que el enunciado se ejecuta, debe repetirse el proceso completo porque las asociaciones para *X* y *Máx* pueden cambiar entre dos ejecuciones del enunciado. LISP, SNOBOL4 y APL casi no utilizan las reglas de alcance estático, y por tanto cada referencia a un nombre en estos lenguajes requiere que durante la ejecución se invoque un proceso bastante complejo y costoso de interpretación, el cual primero encuentra la asociación pertinente para el nombre (en su caso) y luego determina el tipo y atributos del objeto de datos o subprograma asociado.

Las reglas de alcance estático permiten llevar a cabo gran parte de este proceso una vez durante la traducción del programa en lugar de hacerlo repetidamente durante la ejecución para casi todas las referencias a nombres en un programa. Por ejemplo, si el enunciado de asignación $X := X + Máx$ aparece en Pascal, y *Máx* se define por medio de una declaración de constante **const** $Máx = 30$ en alguna parte del programa, entonces las reglas de alcance estático de Pascal permiten relacionar la referencia a *Máx* a esta (o alguna otra) declaración durante la traducción. El compilador de Pascal puede determinar entonces que *Máx* siempre tiene el valor 30 cuando este enunciado se ejecuta y puede traducir el enunciado a código ejecutable que simplemente suma 30 a *X*, sin una operación de referencia para el nombre *Máx*. En forma similar, si las reglas de alcance estático de Pascal permiten relacionar la referencia a *X* con la declaración X : *real* en alguna parte del texto del programa, entonces el compilador de Pascal puede efectuar una verificación estática de tipos; es decir, puede determinar que, cuando el enunciado se ejecuta, (1) existirá una asociación que relaciona *X* con un objeto de datos, (2) ese objeto de datos será de tipo real, y (3) su valor será del tipo correcto para servir como argumento ante la operación de adición. El compilador no puede conocer a partir de la declaración la *localidad* del objeto de datos al que *X* hace referencia (puesto que la localidad se determina dinámicamente durante la ejecución y puede ser diferente en distintas ejecuciones del enunciado), ni puede determinar el *valor* de *X* (porque también se determina dinámicamente durante la ejecución). Sin embargo, la verificación estática de tipos hace que la ejecución del programa sea mucho más rápida y también más fiable (puesto que los errores de tipo se detectan para todas las rutas del programa durante la traducción).

Las reglas de alcance estático permiten establecer muchas clases distintas de conexiones entre referencias a nombres y sus declaraciones durante la traducción. Ya se han mencionado dos de ellas: relacionar un nombre de variable con una declaración para la variable y relacionar un nombre de constante con una declaración para una constante. Otras conexiones incluyen relacionar nombres de tipo con declaraciones de tipo, relacionar parámetros formales con especificaciones de parámetros formales, relacionar llamadas de subprograma con declaraciones de subprograma, y relacionar etiquetas de enunciado a las que se hace referencia en enunciados **goto** con etiquetas en enunciados particulares. En cada uno de estos casos, se puede hacer un conjunto distinto de simplificaciones durante la traducción, las cuales hacen más eficiente la ejecución del programa.

Las reglas de alcance estático también son importantes para el programador al leer un programa, porque permiten relacionar un nombre al que se hace referencia en el programa con una declaración para el nombre sin rastrear el curso de la ejecución del programa. Por ejemplo, las reglas de alcance estático de Pascal permiten relacionar una referencia a X en un enunciado con una declaración para X en otra parte del programa sin considerar en absoluto la serie de llamadas de subprograma que conducen del programa principal a la ejecución efectiva del enunciado. En esta forma, las reglas de alcance estático hacen que el programa sea más fácil de entender. Estas reglas desempeñan un papel importante en el diseño e implementación de casi todos los lenguajes de programación, por ejemplo, Ada, C, FORTRAN, Pascal y COBOL.

7.2.3 Estructura de bloques

El concepto de *estructura de bloques*, tal como se encuentra en *lenguajes estructurados en bloques* como Pascal, PL/I y Ada, merece una mención especial. Los lenguajes estructurados en bloques tienen una estructura de programa característica y un conjunto asociado de reglas de alcance estático. Los conceptos tuvieron su origen en el lenguaje ALGOL 60, uno de los primeros lenguajes más importantes, y debido a su elegancia y su efecto sobre la eficiencia de implementación, han sido adoptados en otros lenguajes.

En un lenguaje estructurado en bloques, cada programa o subprograma está organizado como un conjunto de bloques anidados. La característica principal de un bloque es que introduce un nuevo ambiente local de referencia. Un bloque se inicia con un conjunto de declaraciones para nombres (declaraciones de variable, definiciones de tipo, definiciones de constante, etc.), seguidas de un conjunto de enunciados en los cuales se puede hacer referencia a esos nombres. Para simplificar, consideraremos un bloque como equivalente a una declaración de subprograma, aunque la definición exacta de bloque varía de un lenguaje a otro. Las declaraciones que hay en un bloque definen su ambiente local de referencia. Este ambiente local es invariable durante la ejecución de los enunciados que constituyen su cuerpo. En C, hay una estructura de bloques, pero existe únicamente dentro de un solo subprograma. Esto proporciona la capacidad de suministrar nombres no locales sin los gastos de estructura para la activación de subprogramas. Esto se analizará más adelante.

El anidamiento de bloques se consigue permitiendo que la definición de un bloque contenga cabalmente las definiciones de otros bloques. En el nivel más exterior, un programa se compone

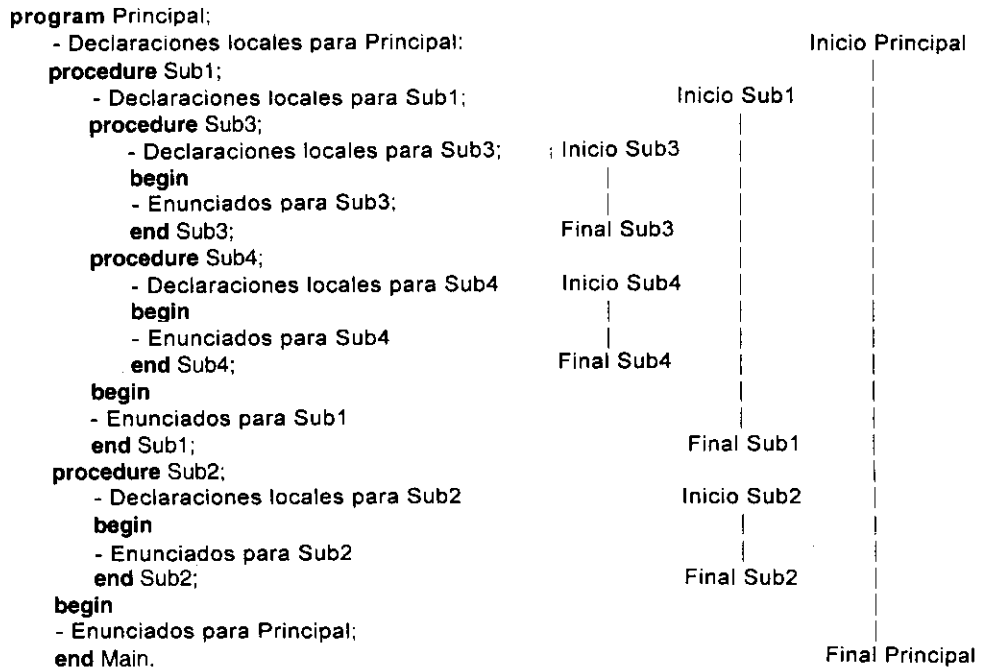


Figura 7.5. Estructura estática de bloques de un programa.

de un bloque único que define el programa principal. Dentro de este bloque hay otros bloques que definen subprogramas que se pueden llamar desde el programa principal; dentro de estos bloques puede haber otros bloques que definen subprogramas invocables desde dentro de los subprogramas del primer nivel, y así sucesivamente. La figura 7.5 ilustra la disposición típica de un programa estructurado en bloques. En lenguajes como C y Ada, el nivel más exterior puede estar compuesto de varios nidos independientes de bloques (cada uno de los cuales se puede compilar por separado), pero aquí bastará con considerar únicamente un nido solo.

Las *reglas de alcance estático* asociadas con un programa estructurado en bloques son como sigue:

1. Las declaraciones que están en la cabeza de cada bloque definen el ambiente local de referencia para el bloque. Cualquier referencia a un identificador dentro del cuerpo del bloque (sin incluir sub-bloques anidados) se considera una referencia a la declaración local para el identificador, si existe uno.

2. Si se hace referencia a un identificador dentro del cuerpo del bloque y no existe una declaración local, entonces la referencia se considera una referencia a una declaración dentro del bloque que encierra inmediatamente al primer bloque. Si no existe una declaración ahí, entonces se refiere a una declaración en el bloque que encierra inmediatamente a ese bloque, y así sucesivamente. Si se alcanza el bloque más exterior antes de encontrar una declaración, entonces la referencia es a la declaración dentro de

ese bloque más externo. Por último, si no se encuentra una declaración ahí, se usa la declaración del ambiente predefinido del lenguaje, en su caso, o la referencia se toma como un error. El ambiente predefinido actúa, por tanto, como un bloque que encierra el bloque (o bloques) más exterior del programa.

3. Si un bloque contiene otra definición de bloque, entonces cualesquier declaraciones locales dentro del bloque interior o de los bloques que el mismo contenga están ocultas por completo del bloque exterior, y no se puede hacer referencia a las mismas desde él. En esta forma, los bloques interiores encapsulan declaraciones para que sean invisibles desde bloques exteriores.

4. Un bloque puede tener nombre (por lo común cuando representa un subprograma con nombre). El nombre del bloque se vuelve parte del ambiente local de referencia del *bloque incluyente*. Por ejemplo, si un programa principal en Pascal contiene una definición de subprograma que comienza así:

   procedure P(A: real);

   entonces el procedimiento llamado *P* es un nombre local en el programa principal, en tanto que el nombre de parámetro formal *A* es parte del ambiente local de *P*. Dentro del programa principal, se puede hacer referencia a *P*, pero no a *A*.

Adviértase que, al usar estas reglas de alcance estático, puede ocurrir una declaración para el mismo identificador en muchos bloques distintos, pero una declaración en un bloque exterior siempre está oculta dentro de un bloque interior si el bloque interior proporciona una nueva declaración para el mismo identificador.

Estas reglas de alcance estático para programas estructurados en bloques permiten que toda referencia a un nombre dentro de cualquier bloque se asocie con una declaración única para el nombre durante la traducción del programa (si la referencia no es un error), con poca acción explícita por parte del programador aparte de suministrar las declaraciones *locales* apropiadas en cada bloque y el anidamiento adecuado de los bloques. El compilador para el lenguaje puede suministrar verificación estática de tipos y otras simplificaciones de la estructura en tiempo de ejecución con base en el uso de las reglas de alcance estático. Por estas razones, la estructura de bloques ha sido adoptada como estructura de programas en muchos lenguajes importantes.

7.2.4 Datos locales y ambientes locales de referencia

Ahora comenzaremos a examinar las diversas estructuras de control de datos que se usan en los lenguajes de programación. Los ambientes locales de referencia, que forman la estructura más sencilla, se tratan en esta sección. Las secciones siguientes consideran los ambientes no locales, así como los parámetros y la transmisión de los mismos.

El *ambiente local* de un subprograma *Q* se compone de los diversos identificadores declarados en la cabeza del subprograma *Q* (pero no *Q* mismo). Los nombres de variables, nombres de parámetros formales y nombres de subprogramas son lo que aquí nos ocupa. Los nombres de subprograma de interés son los nombres de los subprogramas que están definidos localmente dentro de *Q* (es decir, subprogramas cuyas definiciones están anidadas dentro de *Q*).

```
procedure R;
    ...
end;
procedure Q;
var X: integer := 30;    - el valor inicial de X es 30
begin
    write (X);           - imprimir valor de X
    R;                   - llamar el subprograma R
    X := X + 1;          - incrementar valor de X
    write (X)            - imprimir valor de X otra vez
end;
procedure P;
    ...
    Q;                   - llamar el subprograma Q
    ...
end;
```

Figura 7.6. Ambiente local de referencia : ¿Retención o eliminación?

Para ambientes locales, es fácil hacer congruentes las reglas de alcance estático y dinámico. La regla de alcance estático especifica que una referencia a un identificador X en el cuerpo del subprograma Q está relacionada con la declaración local para X en la cabeza del subprograma Q (si se supone que existe una). La regla de alcance dinámico especifica que una referencia a X durante la ejecución de Q se refiere a la asociación para X en la activación actual de Q (adviértase que, en general, puede haber varias activaciones de Q, pero sólo una estará actualmente en ejecución). Para implementar la regla de alcance estático, el compilador mantiene simplemente una tabla de las declaraciones locales para identificadores que aparecen en la cabeza de Q, y mientras compila el cuerpo de Q, recurre primero a esta tabla siempre que se requiere la declaración de un identificador.

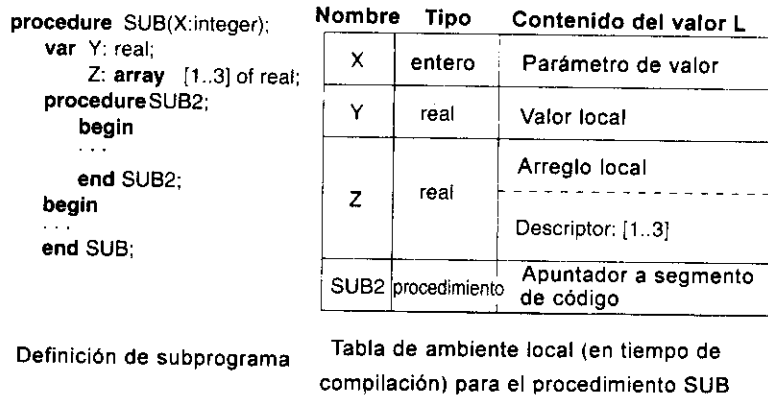
La implementación de la regla de alcance dinámico se puede hacer de dos maneras, y cada una proporciona una semántica distinta a las referencias locales. Considérense los subprogramas P, Q y R, con una variable local X declarada en Q en la figura 7.6. El subprograma P llama a Q, que a su vez llama a R, el cual regresa más tarde el control a Q, mismo que completa su ejecución y regresa el control a P. Sigamos la variable X durante esta secuencia de ejecución:

1. Cuando P está en ejecución, X no es visible en P, puesto que X es local para Q.

2. Cuando P llama a Q, X se hace visible como el nombre de un objeto de datos entero con un valor inicial de 30. Conforme P se ejecuta, el primer enunciado hace referencia a X e imprime su valor actual, 30.

3. Cuando Q llama a R, la asociación para X queda oculta, pero se conserva mientras R se ejecuta.

4. Cuando R regresa el control a Q, la asociación para X se hace visible de nuevo. X todavía nombra el mismo objeto de datos, y ese objeto de datos aún tiene el valor 30.



```
procedure SUB(X:integer);
   var Y: real;
       Z: array [1..3] of real;
   procedure SUB2;
      begin
      . . .
      end SUB2;
   begin
   . . .
   end SUB;
```

Definición de subprograma

Nombre	Tipo	Contenido del valor L
X	entero	Parámetro de valor
Y	real	Valor local
Z	real	Arreglo local
		Descriptor: [1..3]
SUB2	procedimiento	Apuntador a segmento de código

Tabla de ambiente local (en tiempo de compilación) para el procedimiento SUB

Figura 7.7. Tabla de ambiente local en Pascal para el subprograma Sub.

5. Q reanuda su ejecución. Se hace referencia a X y se le incrementa; luego se imprime su nuevo valor, 31.

6. Cuando Q regresa el control a P, la asociación para X vuelve a quedar oculta, pero se pueden proporcionar dos significados distintos para esta asociación:

   Retención. La asociación para X se podría *conservar* hasta que Q sea llamado de nuevo, precisamente como ocurría mientras Q llamaba a R. Si la asociación se conserva, entonces, cuando Q es llamado por segunda vez, Q todavía está asociado con el mismo objeto de datos, el cual tiene aún su valor antiguo, 31. En esta forma, el primer enunciado que se ejecute hará referencia a X e imprimirá el valor 31. Si el ciclo completo se repite y Q es llamado una tercera ocasión, X tendrá el valor 32, y así sucesivamente.

   Eliminación. Alternativamente, la asociación para X se podría *eliminar*; es decir, la asociación que enlaza X al objeto de datos se podría romper, el objeto de datos podría destruirse y su almacenamiento asignarse para algún otro uso. Cuando Q es llamado por segunda vez, se crea un nuevo objeto de datos y se le asigna el valor inicial 30, y también se vuelve a crear la asociación con X. El primer enunciado de Q imprime entonces el valor 30 cada vez que Q se ejecuta.

La retención y la eliminación son dos enfoques distintos de la semántica de ambientes locales y se ocupan del *tiempo de vida* del ambiente. C, Pascal, Ada, LISP, APL y SNOBOL4 utilizan el enfoque de eliminación: las variables locales no conservan sus valores antiguos entre llamadas sucesivas de un subprograma. COBOL y muchas versiones de FORTRAN emplean el enfoque de retención: las variables conservan sus valores antiguos entre llamadas. PL/I y ALGOL proporcionan ambas opciones; cada variable individual se puede tratar de manera diferente.

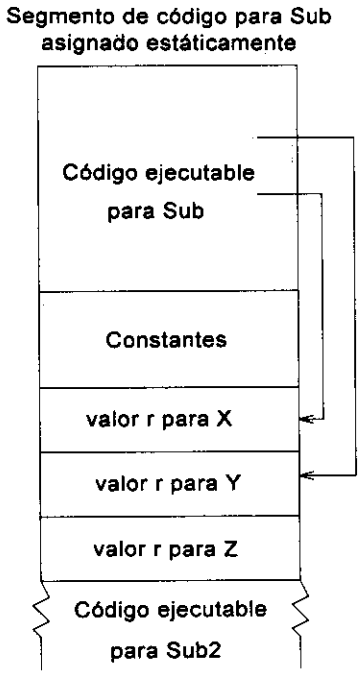


Figura 7.8. Asignación y referencia de variables locales retenidas.

Implementación

Al analizar la implementación de ambientes de referencia, es conveniente representar el ambiente local de un subprograma como una *tabla de ambiente local* compuesta por parejas, donde cada una contiene un identificador y el objeto de datos asociado, como se muestra en la figura 7.7. Según se ha expuesto, el almacenamiento para cada objeto de datos se representa como un tipo (del modo que se comentó en el capítulo 5) y su localidad en la memoria como un valor l (según se expuso en el capítulo 4). El hecho de dibujar una tabla de ambiente local en esta forma no implica que los identificadores mismos (por ejemplo, "X") se guardan durante la ejecución del programa. Ordinariamente no es así. El nombre se usa sólo para que las referencias posteriores a esa variable puedan determinar dónde va a residir esa variable en la memoria durante la ejecución. A través del uso de tablas de ambiente local, la implementación de los enfoques de retención y eliminación hacia los ambientes locales es sencilla.

Retención. Si el ambiente local del subprograma *Sub* de la figura 7.7 se va a conservar entre llamadas, entonces se asigna una sola tabla de ambiente local, que contiene las variables retenidas, como parte del *segmento de código* de *Sub*, como se muestra en la figura 7.8. Puesto que se asigna almacenamiento de manera estática al segmento de código y el mismo continúa existiendo a lo largo de la ejecución, también se retienen todas las variables de la parte de ambiente local del segmento de código. Si una variable tiene un valor inicial, como

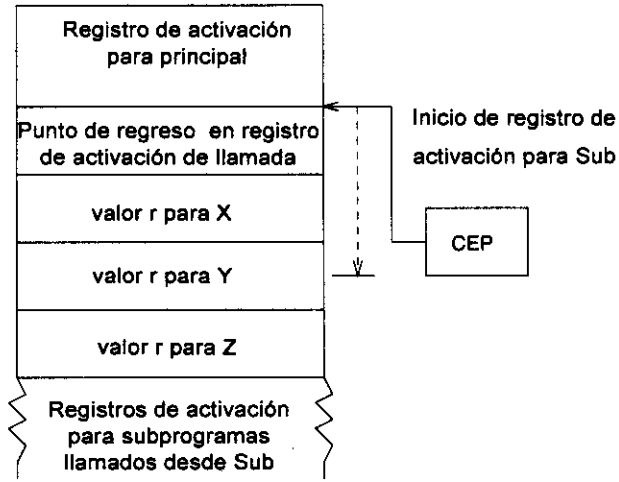


Figura 7.9. Asignación y referencia de variables locales eliminadas.

el valor 30 para *Y* anteriormente, entonces el valor inicial se puede guardar en el objeto de datos cuando se asigna el almacenamiento (precisamente como se guardaría el valor para una constante en el segmento de código). Si se supone que cada variable local retenida se declara al principio de la definición de *Sub*, el compilador puede determinar el tamaño de cada variable en la tabla de ambiente local y calcular el desplazamiento del principio del objeto de datos respecto al inicio (la dirección base) del segmento de código. Cuando un enunciado dentro del código hace referencia a una variable *Y* durante la ejecución, el desplazamiento de *Y* se suma a la dirección base del segmento de código para encontrar la localidad del objeto de datos asociado con *Y*. El identificador "Y" no es necesario durante la ejecución y no se guarda.

Con esta implementación de retención para el ambiente local, no se requiere una acción especial para conservar los valores de los objetos de datos; los valores guardados al final de una llamada de *Sub* seguirán ahí cuando se inicie la llamada siguiente. Tampoco se necesita una acción especial para cambiar de un ambiente local a otro cuando un subprograma llama a otro. Puesto que el código y los datos locales para cada subprograma son parte del mismo segmento de código, una transferencia del control al código para otro subprograma también da por resultado automáticamente una transferencia al ambiente local para ese subprograma.

Eliminación. Si el ambiente local de *Sub* se va a eliminar entre llamadas y a crearse de nuevo en cada entrada, entonces, a la tabla de ambiente local que contiene las variables eliminadas se le asigna almacenamiento como parte del registro de activación para *Sub*. Si se supone que el registro de activación se crea sobre una pila central al entrar a *Sub*, y se elimina de la pila al salir, como se expuso en la sección 7.1, la eliminación del ambiente local se sigue de manera automática. Si se supone que cada variable local eliminada se declara al principio de la definición de *Sub*, el compilador puede determinar de nuevo el número de variables y el tamaño de cada una en la tabla de ambiente local, y puede calcular el desplazamiento del inicio de cada objeto de datos respecto al principio del registro de activación (la dirección

base). Recuérdese que el apuntador CEP (apuntador de ambiente actual) se mantiene durante la ejecución para que apunte a la dirección base del registro de activación en la pila que corresponde al subprograma que se está ejecutando actualmente en cualquier punto. Si *Sub* se está ejecutando y hace referencia a la variable *Y*, entonces la localidad del objeto de datos asociado con *Y* se encuentra sumando el desplazamiento para *Y* al contenido del CEP. Una vez más, no es necesario guardar el identificador "Y" en el registro de activación; sólo se requieren los objetos de datos. La figura 7.9 muestra esta implementación. La flecha punteada muestra el desplazamiento calculado para una referencia a *Y*.

Es fácil implementar ambos enfoques de retención y eliminación en la implementación de nuestro modelo general de subprograma, según se presentó en la sección 6.4.2. Merecen atención varios puntos adicionales:

1. En la implementación de la estructura *simple* de llamada-regreso que se describió anteriormente, la retención y eliminación en ausencia de recursión conduce, esencialmente, a la misma implementación, puesto que nunca hay más de un registro de activación, el cual se puede asigna r de manera estática. Sin embargo, si se suministran valores iniciales para las variables, se obtienen dos significados distintos para la inicialización (véase el problema 2).

2. A las *variables individuales* se les puede dar un tratamiento u otro; a aquéllas cuyos valores se van a retener se les asigna almacenamiento en el segmento de código, y a las otras cuyos valores se van a eliminar se les asigna almacenamiento en el registro de activación. Éste es el enfoque que se emplea en PL/I, donde una variable declarada STATIC (ESTÁTICA) se conserva y una variable declarada AUTOMATIC (AUTOMÁTICA) se elimina.

3. Un *nombre de subprograma*, que está asociado con una declaración para el subprograma en el ambiente local, se puede tratar siempre como retenido. La asociación de nombre y definición se puede representar como un objeto de datos apuntador en el segmento de código, donde el apuntador señala al segmento de código que representa el subprograma.

4. Un *nombre de parámetro formal* representa un objeto de datos que se vuelve a inicializar con un nuevo valor en cada llamada del subprograma, como se describe en la sección 7.3.1, más adelante. Esta nueva inicialización impide conservar un valor antiguo para un parámetro formal entre llamadas. Así pues, es apropiado tratar los parámetros formales como asociaciones eliminadas.

5. Si se permiten llamadas *recursivas* de subprogramas, entonces pueden existir al mismo tiempo múltiples activaciones de un subprograma *Sub* como registros independientes de activación en la pila central. Si una variable *Y* se trata como eliminada, entonces cada registro de activación contendrá un objeto de datos individual llamado *Y*, y, conforme cada activación se ejecuta, hará referencia a su propia copia local de *Y*. Ordinariamente, las copias individuales en cada activación son lo que se desea; por tanto, para los lenguajes con recursión (u otras estructuras de control de subprograma que generan múltiples activaciones simultáneas de un subprograma) se usa ordinariamente

la eliminación de ambientes locales. Sin embargo, la retención de ciertas variables locales suele ser valiosa. En ALGOL 60, por ejemplo, una variable local declarada como **own** (propia) se trata como variable retenida, no obstante que el subprograma que la contiene pueda ser recursivo. Si múltiples activaciones de un subprograma *Sub* hacen referencia a la variable retenida *Y*, entonces existe un solo objeto de datos *Y* que es utilizado por todas las activaciones de *Sub*, y su valor persiste de una activación a otra.

Ventajas y desventajas. Tanto la retención como la eliminación se emplean en un número considerable de lenguajes importantes. El enfoque de retención permite al programador escribir subprogramas que son *sensibles al historial*, en cuanto a que sus resultados en cada llamada están determinados en parte por sus entradas y en parte por los valores de datos locales calculados durante activaciones previas. El enfoque de eliminación no permite acarrear datos locales de una llamada a la siguiente, de modo que una variable que se debe retener entre llamadas se debe declarar como no local para el subprograma. Para subprogramas recursivos, sin embargo, la eliminación es la estrategia más natural. La eliminación también proporciona un ahorro en espacio de almacenamiento en razón de que las tablas de ambiente local existen sólo para aquellos subprogramas que están en ejecución o en ejecución suspendida. Las tablas de ambiente local para todos los subprogramas existen a lo largo de la ejecución cuando se emplea retención.

7.3 DATOS COMPARTIDOS EN SUBPROGRAMAS

Un objeto de datos que es estrictamente local es usado por operaciones sólo dentro de un único ambiente local de referencia, por ejemplo, dentro de un solo subprograma. Los objetos de datos, sin embargo, suelen *compartirse* entre varios subprogramas para que las operaciones de cada uno de los subprogramas puedan utilizar los datos. Un objeto de datos se puede transmitir como un parámetro explícito entre subprogramas, según se explicó en la sección 7.3.1, pero son numerosas las ocasiones donde el uso de parámetros explícitos es engorroso. Por ejemplo, considérese un conjunto de subprogramas en el cual todos hacen uso de una tabla de datos común. Cada subprograma requiere acceso a la tabla, pero transmitir la tabla como un parámetro explícito cada vez que se la necesita es tedioso. Este uso común de datos se basa ordinariamente en la compartición de asociaciones de identificador. Si los subprogramas *P*, *Q* y *R* necesitan todos ellos acceso a la misma variable *X*, entonces es apropiado permitir simplemente que el identificador *X* tenga la misma asociación en cada subprograma. La asociación para *X* se vuelve parte del ambiente local para uno de los subprogramas, y se convierte en parte común del ambiente *no local* de los otros. La compartición de objetos de datos a través de ambientes no locales es una alternativa importante al uso de la compartición directa a través de la transmisión de parámetros.

Se usan cuatro enfoques básicos en cuanto a ambientes no locales en los lenguajes de programación: (1) ambientes comunes explícitos y ambientes no locales implícitos con base en (2) alcance dinámico, (3) alcance estático y (4) herencia. En las subsecciones siguientes se analizará la transmisión de parámetros y tres de estos métodos de ambientes no locales. En el próximo capítulo se trata por separado el concepto de herencia.

7.3.1 Parámetros y transmisión de parámetros

Los parámetros y resultados transmitidos de manera explícita constituyen el método alternativo principal para compartir objetos de datos entre subprogramas. En contraste con el uso de ambientes no locales de referencia, donde la compartición se consigue haciendo visibles ciertos nombres no locales para un subprograma, los objetos de datos transmitidos como parámetros y resultados se transmiten sin un nombre anexo. En el subprograma receptor, a cada objeto de datos se le da un nuevo *nombre local* a través del cual se puede hacer referencia a él. La compartición de datos a través de parámetros resulta más útil cuando a un subprograma se le van a entregar datos distintos para procesar cada vez que se le llama. La compartición a través de un ambiente no local es más apropiada cuando se usan los mismos objetos de datos en cada llamada. Por ejemplo, si el subprograma *P* se usa en cada llamada para introducir un nuevo elemento de datos en una tabla que se comparte con otros subprogramas, entonces la tabla se compartiría típicamente a través de referencias a un ambiente no local y el elemento de datos mismo se transmitiría como un parámetro explícito en cada llamada de *P*.

Parámetros reales y formales

Consideraremos primero la compartición de *datos* a través de parámetros, y más adelante el uso de parámetros para transmitir *subprogramas* y *etiquetas de enunciado*. En los capítulos anteriores, el término *argumento* se usa para un objeto de datos (o un valor) que se envía a un subprograma u operación primitiva como uno de sus operandos, es decir, como un dato para usarse en el procesamiento. El término *resultado* se refiere a un dato (objeto de datos o valor) que es regresado desde una operación al final de su ejecución. El argumento de un subprograma se puede obtener tanto a través de parámetros como a través de referencias no locales (y menos comúnmente a través de archivos externos). En forma similar, los resultados de un subprograma se pueden regresar a través de parámetros, a través de asignaciones a variables no locales (o archivos), o a través de valores explícitos de función. Así pues, los términos *argumento* y *resultado* se aplican a datos enviados al subprograma y regresados por el mismo a través de diversos mecanismos de lenguaje. Al concentrar la atención en los parámetros y su transmisión, los términos *parámetro real* y *parámetro formal* se vuelven fundamentales.

Un *parámetro formal* es una clase particular de objeto de datos local dentro de un subprograma. La definición del subprograma enumera ordinariamente los nombres y declaraciones para parámetros formales como parte de la sección de especificación (encabezamiento). Un nombre de parámetro formal es un simple identificador, y la declaración proporciona ordinariamente el tipo y otros atributos, como en el caso de una declaración de variable local ordinaria. Por ejemplo, el encabezamiento de procedimiento en C:

```
SUB(int X; char Y);
```

define dos parámetros formales llamados *X* y *Y* y declara el tipo de cada uno. Sin embargo, la declaración de un parámetro formal no significa lo mismo que una declaración para una variable. El parámetro formal, de acuerdo con el mecanismo de transmisión que se analizará en

breve, puede ser un seudónimo del objeto de datos de parámetro real o puede contener simplemente una copia del valor de ese objeto de datos.

Un *parámetro real* es un objeto de datos que se comparte con el subprograma de llamada. Un parámetro real puede ser un objeto de datos local perteneciente al que llama, puede ser un parámetro formal del que llama, puede ser un objeto de datos no local visible para el que llama, o puede ser un resultado regresado por una función invocada por el que llama y transmitido de inmediato al subprograma llamado. Un parámetro real se representa en el punto de llamada del subprograma por medio de una expresión, llamada *expresión de parámetro real*, que ordinariamente tiene la misma forma que cualquier otra expresión en el lenguaje (por ejemplo, como una expresión que podría aparecer en un enunciado de asignación). Por ejemplo, el subprograma *SUB* antes especificado podría ser llamado con cualquiera de los tipos siguientes de expresiones de parámetro real:

Llamada de procedimiento en P	Parámetro real en P
SUB	I, B: variables locales de P
SUB	27, true (cierto): constantes
SUB	P1, P2: parámetros formales de P
SUB	G1, G2: variables globales o no locales en P
SUB	Componentes de arreglos y registros
SUB	Resultados de funciones primitivas o definidas

La sintaxis para las llamadas de procedimiento en C es representativa de muchos lenguajes. La llamada de subprograma se escribe en forma prefija, como se expuso en la sección 6.2, con el nombre del subprograma primero, seguido de una lista de expresiones de parámetro real entre paréntesis (sin embargo, se usan otras notaciones, por ejemplo, notación infija en APL, polaca de Cambridge en LISP). Para simplificar, se adoptará la representación prefija convencional de C y se hablará de *listas de parámetros reales* y *listas de parámetros formales* para indicar la serie de parámetros reales y formales designados en una llamada de subprograma y en una definición de subprograma, respectivamente.

Cuando se llama un subprograma con una expresión de parámetro real de cualquiera de las formas anteriores, la expresión se *evalúa* en el momento de la llamada, antes de entrar al subprograma. Los objetos de datos que son resultado de la evaluación de las expresiones de parámetro real se convierten entonces en los parámetros reales que se transmiten al subprograma. El caso especial donde las expresiones de parámetro real *no* se evalúan en el momento de la llamada, sino que se pasan *sin evaluar* al subprograma, se trata por separado más adelante.

Establecimiento de la correspondencia

Cuando se llama un subprograma con una lista de parámetros reales, se debe establecer una correspondencia entre los parámetros reales y los parámetros formales enumerados en la definición del subprograma. Dos métodos son los siguientes:

Correspondencia de posición. La correspondencia se establece apareando parámetros reales y formales con base en sus posiciones respectivas en las listas de parámetros reales y

formales; se aparean el primer parámetro real y el primero formal, luego el segundo de cada lista, y así sucesivamente.

Correspondencia por nombre explícito. En Ada y en algunos otros lenguajes, el parámetro formal que se va a aparear con cada parámetro real se puede nombrar de manera explícita en el enunciado de llamada. Por ejemplo, en Ada, se podría llamar a *Sub* con el enunciado:

```
Sub(Y => B, X => 27);
```

el cual aparea el parámetro real *B* con el parámetro formal *Y* y el parámetro real 27 con el parámetro formal *X* durante la llamada de *Sub*.

Casi todos los lenguajes usan correspondencia de posición de manera exclusiva, de modo que los ejemplos emplean aquí este método. Ordinariamente, el número de parámetros reales y formales debe corresponder para que el apareamiento sea único; sin embargo, ciertos lenguajes relajan esta restricción y suministran convenciones especiales para interpretar parámetros reales faltantes o adicionales. Para simplificar, aquí se supone que todos los parámetros están apareados.

Métodos para transmitir parámetros

Cuando un subprograma transfiere el control a otro subprograma, debe haber una asociación del parámetro real del subprograma que llama con el parámetro formal del programa llamado. Suelen emplearse dos enfoques: se puede evaluar el parámetro real y pasar ese valor al parámetro formal, o el objeto de datos real puede pasarse al parámetro formal. Describiremos esto a través de un proceso en dos pasos:

1. Describir los detalles de implementación del mecanismo de transmisión de parámetros.
2. Describir la semántica de cómo se habrán de usar los parámetros.

Se han ideado en general varios métodos para pasar parámetros reales como parámetros formales. Los primeros cuatro que se describen en seguida son los más comunes: llamada por nombre, llamada por referencia, llamada por valor y llamada por valor-resultado.

Llamada por nombre. Este modelo de transmisión de parámetros contempla una llamada de subprograma como una sustitución del cuerpo completo del subprograma. Con esta interpretación, cada parámetro formal representa la evaluación real del parámetro real particular. Precisamente como si se hicieran las sustituciones reales, cada referencia a un parámetro formal requiere una reevaluación del parámetro real correspondiente.

Esto requiere que en el punto de llamada del subprograma no se evalúen los parámetros reales sino hasta que efectivamente se haga referencia a ellos en el subprograma. Los parámetros se transmiten sin evaluar, y el subprograma llamado determina cuando, en su caso, se evalúan efectivamente. Recuérdese, del análisis previo de las reglas de evaluación uniforme, que esta posibilidad era útil para tratar operaciones como la condicional *if-then-else* como operaciones ordinarias. En operaciones primitivas, la técnica resulta útil en ocasiones; en

subprogramas definidos por el programador su utilidad es más problemática a causa del costo de implementación. La transmisión de parámetros por nombre desempeña un papel importante en ALGOL y es de considerable importancia teórica, pero tiene un costo importante de estructura de ejecución y por ello no se considera como un método popular en cuanto a su uso.

La regla básica de llamada por nombre se puede enunciar en términos de sustitución: el parámetro formal del cuerpo del programa llamado se debe sustituir en todas partes por el parámetro real antes de que se inicie la ejecución del subprograma. Aunque esto parece sencillo, considérese el problema de incluso la sencilla **call** *Sub*(*X*). Si el parámetro formal de *Sub* es *Y*, entonces *Y* se va a sustituir por *X* en todo *Sub* antes de que *Sub* se ejecute. Pero esto no basta, porque cuando se llega a una referencia a *X* durante la ejecución de *Sub*, la asociación para *X* a la que se hace referencia es la que está *en el programa que llama*, no la asociación en *Sub* (en su caso). Cuando *Y* se sustituye por *X*, también se debe indicar un ambiente de referencia distinto para usarse en la referencia de *X*. También se puede introducir ambigüedad si *X* ya es una variable conocida para *Sub*.

No es sorprendente que la técnica básica para implementar la llamada por nombre sea tratar los parámetros reales como subprogramas simples sin parámetros (tradicionalmente llamados en inglés *thunks*). Siempre que se hace referencia en un subprograma a un parámetro formal que corresponde a un parámetro real por nombre, se ejecuta el thunk compilado para ese parámetro, lo que da por resultado la evaluación del parámetro real en el ambiente de referencia apropiado y el regreso del valor resultante como valor del thunk.

Llamada por referencia. La llamada por referencia es tal vez el mecanismo más común de transmisión de parámetros. Transmitir un objeto de datos como parámetro de llamada por referencia significa que se pone a disposición del subprograma un *apuntador* a la localidad del objeto de datos (es decir, su valor l). *El objeto de datos mismo no cambia de posición en la memoria.* Al comienzo de la ejecución del subprograma, los valores l de los parámetros reales se usan para inicializar localidades locales de almacenamiento para los parámetros formales.

El paso de parámetros de llamada por referencia tiene lugar en dos etapas:

1. En el programa de llamada, cada expresión de parámetro real se evalúa para dar un apuntador al objeto de datos del parámetro real (es decir, su valor l). Se guarda una lista de estos apuntadores en un área de almacenamiento común que también es accesible para el programa que se está llamando (a menudo en un conjunto de registros internos o en la pila del programa). El control se transfiere luego al subprograma, como se describe en el capítulo anterior; es decir, se crea el registro de activación para el subprograma (si es necesario), se establece el punto de regreso, y así sucesivamente.
2. En el subprograma llamado, se tiene acceso a la lista de apuntadores a parámetros reales para recuperar los valores r apropiados de los parámetros reales.

Durante la ejecución del subprograma, las referencias a nombres de parámetros formales se tratan como referencias ordinarias a variables locales (excepto en casos que pudiera haber una selección de apuntador oculta; véase más adelante). Al terminar el subprograma, los resultados son regresados al programa de llamada también a través de los objetos de datos de parámetro real.

Llamada por valor. Si un parámetro se *transmite por valor* (se llama *por copia* en Ada 95), el valor (es decir, el valor r) del parámetro real se pasa al parámetro formal llamado. El mecanismo de implementación es similar al modelo de llamada por referencia, excepto que:

- Al invocar un subprograma, un parámetro de llamada por referencia pasa su valor l, en tanto que un parámetro de llamada por valor pasa su valor r.
- Cuando se hace la referencia en el subprograma, un parámetro de llamada por referencia utiliza el valor l guardado en el parámetro formal para tener acceso al objeto de datos real, mientras que, en la llamada por valor, el parámetro formal contiene el valor que se usa.

Con base en este análisis, debe quedar claro que con la llamada por referencia se tiene un sinónimo del parámetro real, en tanto que en la llamada por valor no se tiene esta clase de referencia. Una vez que un parámetro real se pasa por valor, el parámetro formal no tiene acceso a modificar el valor del parámetro real. Todos los cambios hechos a los valores del parámetro formal durante la ejecución del subprograma se pierden cuando el subprograma termina.

Llamada por valor-resultado. Si un parámetro se transmite por *valor-resultado*, el parámetro formal es una variable (objeto de datos) local del mismo tipo de datos que el parámetro real. El valor (es decir, el valor r) del parámetro real se copia en el objeto de datos de parámetro formal en el momento de la llamada, de modo que el efecto es el mismo que si se ejecutara una asignación explícita de parámetro real a parámetro formal. Durante la ejecución del subprograma, todas las referencias al nombre del parámetro formal se tratan igual que una referencia ordinaria a una variable local, como en el caso de la llamada por valor. Cuando el subprograma termina, el contenido final del objeto de datos de parámetro formal se copia en el objeto de datos de parámetro real, de manera similar que si se ejecutara una asignación explícita de parámetro formal a parámetro real. En esta forma, el parámetro real conserva su valor original hasta la terminación del subprograma, cuando se asigna su nuevo valor como resultado del subprograma.

La transmisión de parámetros de valor-resultado se desarrolló a partir del Algol-W, un lenguaje desarrollado por Nicklaus Wirth como sucesor del ALGOL durante los años sesenta, antes de que desarrollara el Pascal. Implementada en una computadora IBM 360, la llamada por referencia era relativamente ineficiente, puesto que no había manera de tener acceso al valor r de un parámetro real en una instrucción. No había una operación de "carga indirecta de memoria". Para acelerar la ejecución, la transmisión de valor-resultado hacía de todos los parámetros "variables locales" directamente direccionables por el apuntador del registro de activación actual.

Tanto la llamada por referencia como por valor-resultado se usan ampliamente y, en casi todas las situaciones "normales", dan el mismo resultado. Sin embargo, las cuatro técnicas, llamada por nombre, llamada por referencia, llamada por valor y llamada por valor-resultado, tienen diferencias que pueden conducir a que programas similares tengan semánticas diferentes, como se muestra más adelante.

Llamada por valor constante. Si un parámetro se *transmite por valor constante*, entonces no se permite cambio alguno en el valor del parámetro formal durante la ejecución del programa;

es decir, no se permite asignar un nuevo valor u otra modificación del valor del pa-rámetro, y el parámetro formal no se puede transmitir a otro subprograma excepto como parámetro de valor constante. El parámetro formal actúa así como una *constante local* durante la ejecución del subprograma. Puesto que ningún cambio en cuanto a su valor es permisible, hay dos implementaciones posibles. El parámetro formal se puede tratar exactamente como un parámetro transmitido por valor para que sea un objeto de datos local cuyo valor inicial es una copia del valor del parámetro real. Alternativamente, se le puede tratar como un parámetro transmitido por referencia de modo que el parámetro formal contenga un apuntador al objeto de datos de parámetro real.

Tanto la llamada por valor como la llamada por valor constante tienen el efecto de proteger al programa que llama contra cambios en el parámetro real. Así pues, desde el punto de vista del programa que llama, el parámetro real es sólo un argumento de entrada para el subprograma. Su valor no puede ser modificado por el subprograma, ya sea inadvertidamente o para transmitir los resultados de regreso.

Llamada por resultado. Un parámetro *transmitido por resultado* se usa sólo para transmitir un resultado de regreso desde un subprograma. El valor inicial del objeto de datos de parámetro real no importa y el subprograma no lo puede usar. El parámetro formal es una variable (objeto de datos) local sin valor inicial (o con la inicialización usual que se proporciona para variables locales). Cuando el subprograma termina, el valor final del parámetro formal se asigna como el nuevo valor del parámetro real, justamente como en la llamada por valor-resultado.

Casi todos los lenguajes implementan uno o dos de los mecanismos anteriores. FORTRAN implementa llamada por referencia, en tanto que Pascal implementa tanto la llamada por valor como la llamada por referencia. Un parámetro *X* : *entero* es un parámetro de llamada por valor, mientras que **var** *X*: *entero* es una llamada por parámetro de referencia. (Adviértase, sin embargo, que esto es una fuente de muchos errores en Pascal. Cuando se olvida incluir la palabra clave **var**, el parámetro es un parámetro de valor y todo cambio que se haga a su valor no se reflejará de regreso en el subprograma que llama. Este error es sutil y extremadamente difícil de encontrar a veces.) Por otra parte, C sólo implementa llamada por valor. Sin embargo, el uso de apuntadores permite al programador construir argumentos de llamada por referencia. El paso de un argumento *i* por referencia al procedimiento *misubrutina* es a través de la expresión *&i*, la cual pasa el valor 1 de *i*. La llamada se escribirá como *misubrutina* (*&i*) a un procedimiento declarado como *miprocedimiento*(int * *x*), el cual declara *x* como un apuntador a un entero. Si se olvida incluir los operadores apropiados para derreferenciar (& y *) puede conducir a muchos errores de programación en C.

Semántica de transmisión

La exposición anterior obliga al programador a estar consciente de cómo se implementan en efecto los parámetros antes de decidir cuál modo de transmisión de parámetros va a usar. Sin embargo, esto va en contra de muchos criterios de diseño de lenguajes donde el programador no se debe preocupar demasiado acerca de los detalles de implementación subyacentes. En Ada, por ejemplo, se emplea un enfoque distinto. En vez de describir el modo de transmisión, se especifica el *papel* del parámetro. El parámetro puede ser un parámetro **in** cuyo valor se

pasa hacia adentro desde el parámetro real al parámetro formal y luego se usa en el subprograma; el parámetro puede ser un parámetro **out** cuyo valor es generado por el subprograma y luego se pasa de regreso al parámetro real al salir del subprograma, o puede ser un parámetro **in out** cuyo valor se pasa hacia adentro y luego el valor resultante se pasa de regreso al parámetro real. Ada está definido de tal manera que el implementador puede elegir métodos alternativos de transmisión entre algunos de éstos. Para el programador, el método particular que se use no influye en el resultado final siempre y cuando el subprograma llamado termine de manera normal y no pueda tener acceso al parámetro real a través de un seudónimo (véase más adelante).

En la definición original de Ada (Ada 83), el implementador del lenguaje tenía la opción de elegir entre llamada por referencia y llamada por valor para parámetros **in** y **out**. Esto causaba problemas de conformidad, puesto que los programas *correctos* podían dar resultados distintos cuando eran compilados por compiladores *correctos* diferentes. Por esta razón, en la revisión Ada 95 al lenguaje, se revisó la transmisión de parámetros:

Los *tipos elementales de datos* (por ejemplo, escalares como entero, real o booleano) se pasan por valor constante si se trata de un parámetro **in**, o por valor-resultado si el parámetro es **out** o **in out**.

Los *tipos de datos compuestos* (por ejemplo, arreglos y registros) se pasan por referencia.

Esto ha simplificado la transmisión de parámetros en Ada, pero deja todavía la oportunidad de una ejecución confusa, como lo demuestra el problema 5 al final del capítulo.

Valores de función explícitos

En casi todos los lenguajes, un solo resultado se puede regresar como *valor de función* explícito antes que como parámetro. El subprograma se debe declarar como un subprograma de función, y el tipo de resultado regresado se debe declarar como parte de la especificación del subprograma, como en la declaración en C: *float fn(int a)*, la cual especifica que *fn* es un subprograma de función que regresa un resultado de tipo *real*. Dentro del subprograma de función, el resultado por regresar como valor de la función se puede especificar en una de dos formas. Un método, que se usa en C, consiste en designar el valor de función por devolver por medio de una expresión de resultado explícita dada como parte del enunciado **return** que termina la ejecución del subprograma, por ejemplo, **return** $2 * x$ para designar que el valor $2 * x$ debe regresarse como valor de la función. Un método alternativo, que se usa en Pascal, consiste en designar el valor por devolver por medio de una asignación del valor al *nombre de la función* dentro del subprograma, por ejemplo, *fn* := $2 * x$. En este último método, el subprograma puede contener varias asignaciones al nombre de la función. El valor de función regresado es el último valor asignado al nombre de la función antes de que el subprograma termine. Con cualquiera de los métodos, es mejor considerar el valor de la función como un parámetro "out" adicional implícito proveniente del subprograma.

Implementación de la transmisión de parámetros

Puesto que cada activación de un subprograma recibe un conjunto distinto de parámetros, el almacenamiento para los parámetros formales de un subprograma se asigna ordinariamente

como parte del registro de activación del subprograma, en vez de en el segmento de código. Cada parámetro formal es un objeto de datos local en el subprograma. Si un parámetro formal *P* se especifica en el encabezamiento del subprograma como de un tipo particular *T* (es decir, el parámetro real transmitido es del tipo *T*), entonces el parámetro formal se implementa en una de dos formas, según el método de transmisión de parámetros que se esté usando (como ya se ha explicado). O bien *P* se trata como un objeto de datos local de tipo *T* (cuyo valor inicial puede ser una copia del valor del parámetro real), o *P* se trata como un objeto de datos local de tipo *apuntador a T* (cuyo valor inicial es un apuntador al objeto de datos de parámetro real). El primer método se usa para parámetros transmitidos por valor-resultado, por valor y por resultado; el segundo se usa para parámetros transmitidos por referencia. Cualquiera de los métodos se puede usar para implementar parámetros transmitidos por valor constante. Un valor de función explícito se puede tratar con el primer método. Si el lenguaje no suministra una especificación de tipo para parámetros formales (como en LISP, APL y SNOBOL4), entonces el parámetro formal se puede implementar como una variable de apuntador local, aunque el apuntador puede señalar hacia un objeto de datos de tipo arbitrario.

Las diversas acciones asociadas con la transmisión de parámetros se dividen en dos grupos, las asociadas con el *punto de llamada* del subprograma en cada subprograma que llama, y las asociadas con la *entrada y salida* en el subprograma mismo. En el punto de llamada, en cada subprograma que llama, se evalúan las expresiones de parámetro real y se establece la lista de apuntadores a objetos de datos de parámetro real (o a veces, simplemente a copias de sus valores). Adviértase que es importante que esta evaluación tenga lugar en el punto de llamada, en el ambiente de referencia del subprograma que llama. Cuando se han determinado los parámetros reales, el control se transfiere al subprograma llamado. Ordinariamente, esto implica un cambio en los apuntadores CIP y CEP, como se explicó en el capítulo 6, para transferir el control al inicio del código ejecutable para el subprograma, y también para cambiar el ambiente de referencia al apropiado para el subprograma llamado.

Después de la transferencia de control al subprograma, el *prólogo* para el subprograma completa las acciones asociadas con la transmisión de parámetros, ya sea copiando el contenido completo del parámetro real en el parámetro formal, o copiando el apuntador al parámetro real en el parámetro formal. Antes de que termine el subprograma, el *epílogo* para el subprograma debe copiar los valores de resultado en los parámetros reales transmitidos por resultado o valor-resultado. Los valores de función también se deben copiar en registros internos o en almacenamiento temporal proporcionado por el programa que llama. El subprograma termina entonces y su registro de activación se pierde, de manera que ordinariamente todos los resultados deben copiarse para sacarlos del registro de activación antes de la terminación.

El compilador tiene dos tareas principales en la implementación de transmisión de parámetros. Primero, debe generar el código ejecutable correcto para transmisión de parámetros, para el regreso de resultados, y para cada referencia a un nombre de parámetro formal. Puesto que casi todos los lenguajes suministran más de un método de transmisión de parámetros, el código ejecutable que se requiere en cada caso suele ser ligeramente distinto. Esta generación de código también es difícil porque suele implicar la adopción de acciones coordinadas en cada punto de llamada, en el prólogo del subprograma y en su epílogo. La segunda tarea principal del compilador es llevar a cabo la verificación estática de tipos necesaria para asegurar que el tipo de cada objeto de datos de parámetro real concuerda con el declarado para el

```
Q(int i, int *j)
    { i=i+10;
    *j=*j+10;
    printf("i=%d; j=%d;\n",i,*j);
     }
```

(a) Subprograma llamado

```
P()
    {int a, b;
    a = 2;
    b = 3;
    Q(a,&b);
    printf("%d %d\n",a,b);
    }
```

(b) Subprograma de llamada

Figura 7.10. Parámetros por valor y por referencia en C.

parámetro formal correspondiente. Para esta verificación, el compilador debe conocer la especificación del subprograma que se está llamando (número, orden y tipo de parámetros) pero no necesita conocer la estructura interna del cuerpo del subprograma. Esta especificación debe estar disponible en cada punto de llamada del subprograma. En muchos lenguajes, en particular aquellos donde los subprogramas se pueden compilar por separado unos de otros, si un subprograma Q es llamado desde un subprograma P, entonces se debe proporcionar una especificación por separado para Q cuando se compila P, no obstante que Q está definido en otra parte, con el fin de permitir al compilador efectuar la verificación estática de tipos necesaria y generar el código apropiado para la transmisión de parámetros en cada punto de llamada.

Ejemplos de transmisión de parámetros

La combinación del método de transmisión de parámetros con distintos tipos de parámetros reales conduce a diversos efectos. Algunos ejemplos servirán para explicar las sutilezas. Para estos ejemplos, las llamadas *por referencia* y *por valor* son los dos métodos que se utilizan. Las diferencias que resultan del uso de otros métodos se tratan en algunos de los problemas. C se usa como base para estos ejemplos. En nuestra versión de C, un nombre de parámetro formal que está precedido por * en el encabezamiento del subprograma se transmite por referencia, en tanto que uno sin * se transmite por valor.

Variables simples y constantes. La figura 7.10(a) muestra un subprograma Q en C con dos parámetros formales, i, transmitidos por valor, y j, que se transmite por referencia. Supóngase que se escribe un subprograma P que llama a Q con dos variables enteras, a y b, como parámetros reales, según se ve en la figura 7.10(b). Si P se ejecuta, los resultados que imprimen los dos enunciados de *escritura* son: 12 13 2 13. Sigamos cada parámetro por turno:

Cuando P llama a Q, se evalúan las expresiones de parámetro real a y $\&b$; es decir, se invoca una operación de referencia para determinar la asociación actual de los nombres a y b. Cada nombre representa un objeto de datos de variable entera, de modo que los parámetros reales que se transmiten son el valor r de a y el valor l de b. Puesto que a se está transmitiendo por valor, el parámetro formal i se representa como una variable local entera dentro de Q, y cuando el subprograma Q inicia su ejecución, el valor de a en el momento de la llamada se

asigna como valor inicial de *i*. En adelante, *a* e *i* no tienen más conexión. Así, cuando a *i* se asigna el nuevo valor 12, *a* no cambia. Después de que se ha completado la llamada a *Q*, *a* tiene todavía el valor 2.

El parámetro *b*, por otra parte, se transmite por referencia. Esto significa que *j* es una variable local en *Q* del tipo *apuntador a entero*. Cuando *Q* inicia su ejecución, el apuntador al objeto de datos *b* se guarda como el valor r de *j*. Cuando se suma 10 al valor de *j*, *j* mismo no cambia. En vez de ello, cada referencia a *j* (el valor r de *j*, que es el valor l de *b*) va seguida de una operación de selección de valor de apuntador, de manera que la referencia a *j* proporciona efectivamente acceso a la localidad del objeto de datos *b*. Como resultado la asignación a *j*, aunque parece similar a la asignación a *i*, significa de hecho algo muy diferente. El valor del parámetro real *b* cambia a 13. Cuando los valores de los parámetros formales *i* y *j* se imprimen en *Q*, los resultados son 12 y 13, pero después de el regreso a *P*, cuando se imprimen los valores de los parámetros reales correspondientes *a* y *b*, sólo *b* ha cambiado de valor. El valor 12 asignado a *i* en *Q* se pierde cuando *Q* termina, puesto que las variables locales de *Q* se eliminan al terminar. El valor de *j*, desde luego, también se pierde, pero se trata del apuntador, no del valor 13.

Estructuras de datos. Supóngase que se escribe una versión diferente de *Q* en la cual los parámetros formales son vectores. Existe un problema para escribir este ejemplo en C, pues este lenguaje no pasa normalmente valores de arreglo por valor. En vez de ello, la declaración de procedimiento *sub*1(*int a*[20]) se interpreta igual que si se escribiera *sub*1(int * *a*). En otras palabras, C interpreta los parámetros de arreglo igual que en Ada 95, pasando un apuntador al arreglo (su valor l) en vez de pasar el valor r (una copia del arreglo). Por tal razón, esta parte del ejemplo se da en Pascal, el cual copia en el subprograma el valor r de un parámetro de arreglo de llamada por valor:

```
type vect = array [1 . . 3] of integer;
procedure Q(k: vect; var l: vect);
   var n: integer;
   begin
      k[2] := k[2] + 10;
      l[2] := l[2] + 10;
      for n := 1 to 3 do write(k[n]);
      for n := 1 to 3 do write(l[n])
   end;
```

Se podría escribir el procedimiento *P* como se da en la figura 7.11.

Cuando *P* se ejecuta, los valores que se imprimen son: 6 17 8 6 17 8 6 7 8 6 17 8

Para seguir la transmisión de *c* y *d* como parámetros, primero se advierte que la evaluación de las expresiones de parámetro real *c* y *d* en *P* conduce los apuntadores hacia bloques de almacenamiento para los vectores *c* y *d* dentro del registro de activación de *P* (igual que para *a* y *b* anteriormente). Estos apuntadores se transmiten a *Q*. Puesto que el vector *c* se transmite por valor, el parámetro formal correspondiente *k* es un arreglo local en *Q* que tiene la misma forma que *c* (tres componentes más un descriptor). Los tres valores del vector *c* se copian en

```
procedure P;
    var c,d: vect;
        m: integer;
    begin
        c[1] := 6; c[2] := 7; c[3] := 8;
        d[1] := 6; d[2] := 7; d[3] := 8;
        Q(c,d);
        for m := 1 to 3 do write(c[m]);
        for m := 1 to 3 do write(d[m])
    end;
```

Figura 7.11. Parámetros de estructura de datos.

las posiciones correspondientes de k, y en adelante c y k no tienen más contacto, de modo que, al devolver a P, el vector c no ha sido modificado por la asignación a k en Q. Sin em-

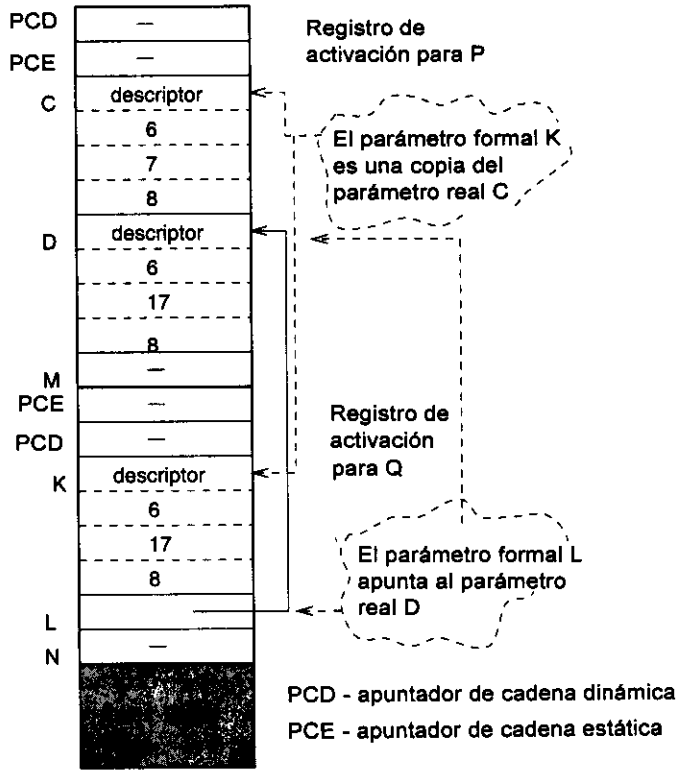


Figura 7.12. Pila en tiempo de ejecución antes de que Q termine.

bargo, el vector *d* transmitido por referencia es modificado por la asignación al vector de parámetro formal *l*, porque *l* no es un vector sino sólo un apuntador a un vector. Cuando *Q* inicia su ejecución, *l* se inicializa para que apunte al vector *d*, y cada referencia subsiguiente a *l* en *Q* conduce a *d* por intermedio de este apuntador. Por tanto, la asignación a *l* [2] modifica a *d* [2]. Los valores impresos reflejan estas distinciones. La figura 7.12 muestra la pila en tiempo de ejecución al final de la ejecución de *Q*.

En general, una estructura de datos como un arreglo o registro que se transmite por valor se copia en una estructura de datos local (el parámetro formal) en el subprograma llamado. El subprograma trabaja sobre esta copia local y no tiene acceso al original. Una estructura de datos transmitida por referencia no se copia, y el subprograma llamado trabaja directamente sobre la estructura de datos de parámetro real (usando el apuntador de parámetro normal para tener acceso).

Componentes de las estructuras de datos. Supóngase que volvemos al procedimiento *Q* de la figura 7.10(a), pero en vez de variables simples o constantes, pasamos componentes de estructuras de datos como parámetros a *Q*. Por ejemplo, escribiendo *P* como:

```
P()
   {int c[4];
   int m;
   c[1] = 6; c[2] = 7; c[3] = 8;
   Q(c[1],&c[2]);
   for (m = 1;m<=3; m++) printf("%d\n",c[m]);
   }
```

Cuando *P* se ejecuta, los valores que se imprimen son: 16 17 6 17 8

La transmisión de *c*[1] por valor sigue el mismo patrón que antes. La expresión *c*[1] se evalúa haciendo referencia a *c* y seleccionando luego su primer componente. El resultado es el valor r de ese componente. El parámetro formal *i* se inicializa a este valor r como antes, de modo que el resto de las acciones son las mismas. De manera similar, se evalúa &*c*[2] y se transmite un apuntador al componente. Las asignaciones dentro de *Q* cambian entonces directamente el componente de *c* a través del apuntador guardado en el parámetro formal *j*.

Los componentes de *c* se representan en almacenamiento *exactamente como objetos de datos de variable simples del mismo tipo*. Dentro de *Q*, el código que se ejecuta para manipular *i* y *j* es el mismo, independientemente de si la llamada de *Q* es *Q*(*a*, *b*) o *Q*(*c*[1],*c*[2]). Si los componentes de *c* se representaran en almacenamiento de manera distinta a las variables simples *a* y *b* (por ejemplo, si estuvieran "empacados" de alguna manera), entonces se tendría que hacer una conversión especial antes de que se llamara a *Q* para convertir los parámetros reales a la representación correcta (según lo que espere *Q*), guardar los parámetros convertidos en almacenamiento temporal y pasar apuntadores al almacenamiento temporal a *Q*. Para llamada por valor, esta conversión podría ser aceptable, pero para llamada por referencia, el apuntador resultante ya no es un apuntador al objeto de datos original, de modo que las asignaciones al parámetro formal ya no modifican directamente el parámetro real. Por esta

razón, la transmisión de componentes de arreglos y registros empacados suele estar prohibida (por ejemplo, en Pascal).

Componentes de arreglo con subíndices computados. Supóngase que el subprograma *R* tiene dos parámetros enteros transmitidos por referencia:

```
R(int *i, int *j)
    {*i = *i + 1;
    *j = *j +1;
    printf("%d %d\n",*i,*j);
    }
```

Supóngase que *P* es como antes, pero la llamada a *Q* se sustituye por *R*:

```
P()
    {int c[4];
    int m;
    c[1] = 6; c[2] = 7; c[3] = 8;
    m = 2;
    R(&m,&c[m]);
    for (m = 1;m<=3; m++) printf("%d\n",c[m]);
    }
```

¿Qué valores se imprimen cuando *P* se ejecuta ahora? Adviértase que *m* tiene inicialmente el valor 2, pero, como es un parámetro de referencia, su valor se cambia a 3 en *R* antes de que *c*[*m*] se incremente a través del apuntador de *j*. ¿Suma entonces el enunciado $*j = *j + 1$ uno a *c*[2] o *c*[3]? Es claro que debe ser *c*[2] lo que se incremente, no *c*[3], porque la expresión de parámetro real *c*[*m*] se evalúa *en el momento de la llamada de R* para obtener un apuntador a un componente de *c*. En el momento de la llamada de *R*, *m* tiene el valor 2, de modo que es un apuntador a *c*[2] que se transmite a *R*. *R* nada sabe de la existencia de *c*[3], puesto que dentro de *R* el apuntador a *c*[2] parece igual que un apuntador a cualquier otro objeto de datos entero. Por tanto, los valores que se imprimen son: 3 8 6 8 8. (Aparece un efecto distinto si los parámetros se transmiten *por nombre*.)

Apuntadores. Supóngase que un parámetro real es una variable simple de tipo *apuntador* o es una estructura de datos como un arreglo o registro que contiene valores l (apuntadore s a objetos de datos) como componentes. Por ejemplo, supóngase que *X* en *P* se declara como:

```
vect *x;
```

y supóngase que el parámetro formal correspondiente en *Q* se declara de manera semejante:

```
vect *h:
```

Independientemente de si *h* se declara como transmitido por valor o por referencia, el efecto de transmitir *x* como el parámetro real es permitir que *Q* tenga acceso directo al vector al cual apunta *x*. Si la transmisión es por valor, entonces *Q* tiene su propia copia del valor l que *x*

contiene, de modo que tanto x como h apuntan al mismo vector. Si la transmisión es por referencia, entonces h contiene un apuntador a x, que contiene a su vez un apuntador al vector. Como regla general, siempre que el objeto de datos de parámetro real contiene un apuntador o componentes apuntadores, los objetos de datos designados por estos apuntadores serán directamente accesibles desde el subprograma, independientemente del método de transmisión de parámetros. Adviértase que si una lista vinculada (u otra estructura de datos vinculada) se transmite *por valor* como parámetro a un subprograma, esto significa por lo común que sólo el apuntador al primer elemento se copia durante la transmisión; la estructura vinculada completa no se copia en el subprograma. Es a causa de esta propiedad de las variables apuntador y de la adición de diversos operadores de valores l y valores r en C que C no necesita un mecanismo explícito de llamada por referencia y la llamada por valor es suficiente.

Resultados de expresiones. Si se desea pasar una expresión por referencia, como la expresión $a + b$ en $Q(\&(a + b),\&b)$, el traductor debe evaluar primero la expresión en el punto de llamada, luego guardar el valor resultante en una *localidad de almacenamiento temporal* en P, y después transmitir un apuntador a esa localidad al procedimiento Q como parámetro. La ejecución de Q prosigue como antes. La transmisión por referencia conduce a que el parámetro formal contenga un apuntador a la localidad de almacenamiento temporal en P. Puesto que esta localidad no tiene un nombre por el cual se pueda hacer referencia a ella en P, cualquier asignación que Q haga a ella no cambia un valor al que P pueda hacer referencia más adelante. Así pues, Q no puede transmitir un resultado de regreso a P a través del parámetro de referencia. Tanto C como Pascal prohíben este caso cuando la transmisión es por referencia, porque las asignaciones al parámetro formal no tienen un efecto observable en el programa de llamada (y por tanto se debe usar un parámetro transmitido por valor).

Seudónimos y parámetros

La posibilidad de seudónimos (nombres múltiples para el mismo objeto de datos en un solo ambiente de referencia) surge en relación con la transmisión de parámetros en casi todos los lenguajes. Como se explicó en la sección 7.2.1, los seudónimos son un problema para entender y verificar la corrección de los programas y en la optimización de los mismos. Se puede crear un seudónimo durante la transmisión de parámetros en una de dos formas:

1. *Parámetro formal y variable no local.* Un objeto de datos transmitido como parámetro real por referencia puede ser directamente accesible en el subprograma a través de un nombre no local. El nombre de parámetro formal y el nombre no local se vuelven entonces seudónimos porque cada uno se refiere al mismo objeto de datos. La figura 7.4 muestra un ejemplo de este tipo de creación de seudónimos.
2. *Dos parámetros formales.* El mismo objeto de datos se puede transmitir como un parámetro real por referencia en dos posiciones de la misma lista de parámetros reales. Los dos nombres de parámetro real se convierten entonces en sinónimos porque se puede hacer referencia al objeto de datos a través de uno u otro nombre. Por ejemplo, el procedimiento anterior R, definido con la especificación en Pascal:

procedimiento R(**var** i,j: integer);

podría ser llamado por P usando *R*(*m*,*m*). Durante la ejecución de *R*, tanto *i* como *j* contienen apuntadores al mismo objeto de datos *m* en *P*; por tanto, *i* y *j* son seudónimos. FORTRAN prohíbe esta forma de creación de seudónimos.

Subprogramas como parámetros

En muchos lenguajes, un subprograma se puede transmitir como un parámetro real a otro subprograma. La expresión de parámetro real consiste en ese caso en el *nombre* del subprograma. El parámetro formal correspondiente se especifica entonces como de tipo *subprograma*. En Pascal, por ejemplo, un subprograma *Q* se puede definir con un parámetro formal *R* de tipo procedimiento o función:

procedure Q(x:integer; **function** R(y,z: integer): integer);

y *Q* se puede llamar entonces con un subprograma de función como su segundo parámetro, por ejemplo, la llamada *Q*(27, *fn*) que invoca *Q* y pasa el subprograma de función *fn* como parámetro. Dentro de *Q*, el subprograma que se pasó como parámetro se puede invocar usando el nombre de parámetro formal, *R*, por ejemplo, *z* := *R*(*i*, *x*) invoca el subprograma de parámetro formal (*fn* en la llamada anterior) con los parámetros reales *i* y *x*; por tanto, *R*(*i*, *x*) es equivalente a *fn*(*i*, *x*) en este caso. En una llamada distinta, si el parámetro real es el subprograma de función *fn*2, entonces *R*(*i*, *x*) invoca a *fn*2(*i*, *x*).

Hay dos problemas importantes asociados con los parámetros de subprograma:

Verificación estática de tipos. Cuando se llama un parámetro de subprograma usando el nombre de parámetro formal, por ejemplo, *R*(*i*, *x*) del caso anterior, es importante que sea posible la verificación estática de tipos para asegurar que la llamada incluya el número y tipo apropiados de parámetros reales para el subprograma que se está llamando. Puesto que el nombre real del subprograma que se está llamando no se conoce en el punto de llamada (en el ejemplo anterior, es *fn* en una llamada y *fn*2 en otra), el compilador no puede determinar ordinariamente si los parámetros reales *i* y *x* de la llamada *R*(*i*, *x*) coinciden con los esperados por *fn* o *fn*2 sin información adicional. Lo que el compilador necesita es una especificación completa para el parámetro formal *R* que incluya no sólo el tipo *procedimiento* o *función*, sino además el número, orden y tipo de cada parámetro (y resultado) de ese procedimiento o función, por ejemplo, como se dieron anteriormente en la especificación para *Q*. Entonces, dentro del subprograma *Q*, cada llamada de *R* se puede verificar de manera estática en cuanto a la corrección de su lista de parámetros. Además, en cada llamada de *Q* el parámetro real que corresponde a *R* se puede verificar en cuanto a su especificación y ver si concuerda con la que se dio para *R*. Así pues, *fn* y *fn*2 deben tener ambas el mismo número, orden y tipo de parámetros que se especificaron para *R*.

Referencias no locales (*variables libres*). Supóngase que un subprograma como *fn* o *fn*2 contiene una referencia a una variable no local. Por ejemplo, supóngase que *fn* hace referencia a *z* y *fn*2 hace referencia a *z*2, y ninguno de los subprogramas contiene una definición local para la variable a la que hace referencia. Una referencia no local de esta clase se suele designar,

```
program Principal
    var X: integer;
    procedure Q(var I:integer; function R(J:integer):integer);
        var X:integer;
        begin
        X := 4;
        write("En Q, antes de llamada de R, I=", I, "X=", X);
        I := R(I);
        write("En Q, después de llamada de R, I=", I, "X=", X)
        end;
    procedure P;
        var I: integer;
        function FN(K:integer):integer;
            begin;
            X := X+K;
            FN := I+K;
            write("En P, I=", I, "K=", K, "X=", X)
            end;
        begin
        I := 2;
        Q(X,FN);
        write("En P, I=", I, "X=", X)
        end;
    begin
    X := 7;
    P;
    write("En Principal, "X=", X)
    end.
```

En FN, I y X son variables libres aquí

Figura 7.13. Variables libres como parámetros de subprograma en Pascal.

en matemáticas, como una *variable libre*, porque no tiene enlaces locales dentro de la definición del subprograma. Ordinariamente, cuando se llama un programa se establece un ambiente no local de referencia, y este ambiente no local se usa durante la ejecución del subprograma para proporcionar un significado para cada referencia a una variable no local (como se describe en las secciones siguientes). Sin embargo, supóngase que un subprograma *fn* que contiene una referencia no local se pasa como parámetro desde un subprograma de llamada *P* a un subprograma llamado *Q*. ¿Cuál ambiente no local deberá usarse cuando *fn* se invoca dentro de *Q* [usando el parámetro formal correspondiente *R*, por ejemplo, *R*(*i*, *x*) para invocar a *fn*(*i*, *x*)]? La respuesta más sencilla consiste en decir que el ambiente no local deberá ser el mismo que el que se usaría si la llamada *R*(*i*, *x*) se reemplazara simplemente por la llamada *fn*(*i*, *x*) en el subprograma *Q*, pero eso resulta ser la respuesta errónea en casi todos los casos. La figura 7.13 ilustra la dificultad. La función *fn* contiene referencias no locales tanto a *x* como a *i*. De acuerdo con las reglas de alcance estático del Pascal, *x* hace referencia al *x* declarado en el programa principal e *i* hace referencia al *i* declarado en el procedimiento *P*. Sin embargo, *P* pasa *fn* como parámetro a *Q*, y es *Q* el que llama en efecto a *fn* a través del nombre de parámetro formal *R*. *Q* tiene definiciones locales tanto para *i* como para *x*, y no se puede permitir que *fn* recupere incorrectamente estas variables locales cuando se le llama.

Este problema de variables libres como funciones que se pasan como parámetros no es exclusivo de lenguajes como Pascal que utilizan reglas de alcance estático y estructura de bloques. También se presenta en LISP y otros lenguajes que usan la regla de la asociación más reciente para referencia no local. La solución general consiste en invocar la regla siguiente acerca del significado de las variables libres en parámetros funcionales: una referencia no local (referencia a una variable libre) deberá significar lo mismo durante la ejecución del subprograma que se pasa como parámetro que lo que significaría *si el programa se invocara en el punto donde aparece como parámetro real en la lista de parámetros*. Por ejemplo, en la figura 7.13 el subprograma *fn* aparece como un parámetro real en la lista de parámetros de la llamada a *Q* dentro de *P*. Por tanto, las referencias no locales a *x* e *i* en *fn*, independientemente de dónde se llame en efecto a *fn* más adelante (en este caso dentro de *Q*) van a significar justo lo que significarían si *fn* fuera llamado donde se llama a *Q* dentro de *P*.

Para implementar correctamente esta regla en cuanto al significado de referencias no locales dentro de subprogramas que se pasan como parámetros, deberá ser posible recrear el ambiente no local correcto en el punto de llamada del parámetro de subprograma para que se ejecute usando el ambiente no local correcto. En la implementación de cadena estática de referencia no local, como se expuso en la sección 7.3.4, esto es bastante sencillo. Todo lo que se necesita es determinar el apuntador de cadena estática correcto para el parámetro de subprograma y pasarlo como parte de la información que se transmite con el parámetro de subprograma. El parámetro de subprograma se convierte entonces en un par de apuntadores (PC, PCE), donde PC es un apuntador al segmento de código para el subprograma y PCE es el apuntador de cadena estática que se debe usar cuando se invoca el subprograma. Esta pareja se puede pasar a través de muchos niveles de subprograma hasta que llega el momento de invocar el subprograma. En ese punto se crea el registro de activación, se instala el PCE transmitido y la ejecución del segmento de código de subprograma tiene lugar como para cualquier otra llamada.

La figura 7.14 ilustra los pasos principales en la ejecución del programa en Pascal de la figura 7.13. Para ilustrar la interacción entre cadenas estáticas (PCE) y dinámicas (PCD), así como los parámetros de subprograma y el ambiente de referencia, el ejemplo muestra los valores de variables, puntos de regreso y apuntadores de cadena estática durante los pasos principales en ejecución del programa. El comportamiento del programa es como sigue:

```
EN Q:          ANTES DE LLAMADA DE R, I=7, X=4
EN FN:         I=2, K=7, X=14
EN Q:          DESPUÉS DE LLAMADA DE R, I=9, X=4
EN P:          I=2, X=9
EN Principal:  X=9
```

Etiquetas de enunciado como parámetros

Muchos lenguajes permiten pasar una etiqueta de enunciado como parámetro a un subprograma y usarla luego como objeto de un enunciado **goto** dentro del subprograma. Además de las dificultades usuales asociadas con el uso de enunciados **goto**, como se describe en el capítulo 6, este mecanismo introduce dos nuevas dificultades:

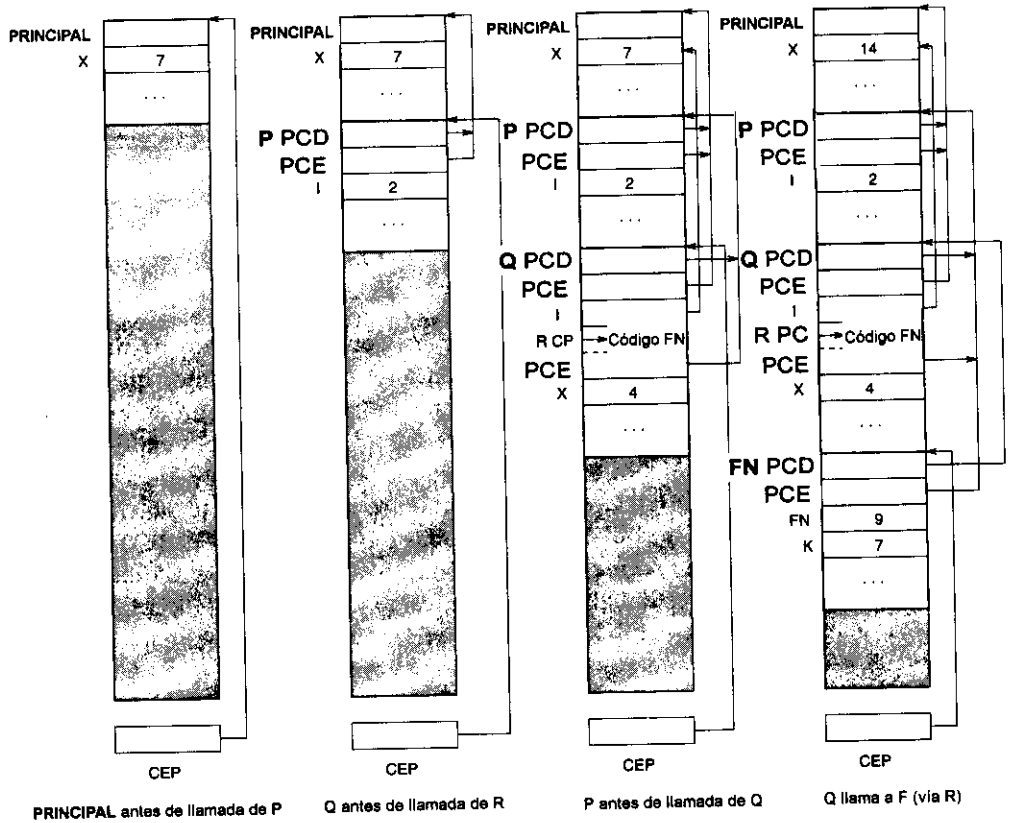


Figura 7.14. Instantáneas de la pila central durante la ejecución de Pascal.

¿Qué activación se debe usar? Una etiqueta de enunciado se refiere a una instrucción particular en el segmento de código de un subprograma durante la ejecución. Sin embargo, un **goto** no puede transferir simplemente el control a esa instrucción cambiando el CIP en la forma usual, porque el segmento de código puede ser compartido por varias activaciones del subprograma. La etiqueta de enunciado, cuando se pasa como parámetro, debe designar una instrucción *en una activación particular* del subprograma. En esta forma, la etiqueta se convierte en una pareja (apuntador de instrucción, apuntador de registro de activación) que se transmite como parámetro.

¿Cómo se implementa el **goto** *a un parámetro de etiqueta?* Cuando se ejecuta un **goto** que designa un parámetro formal de tipo etiqueta como su objeto, en la mayoría de los casos no basta con transferir simplemente el control a la instrucción designada en la activación señalada. En vez de ello, la cadena dinámica de llamadas de subprograma se debe desenvolver hasta que se alcanza la activación de subprograma designada. Esto es, se debe terminar el subprograma actual que está ejecutando el enunciado **goto**, luego el subprograma que lo llamó, luego el que llamó a ese subprograma, y así sucesivamente, hasta que se llega a la activación de subprograma designada en el **goto**. Esa activación comienza entonces a ejecutarse

en la instrucción designada por el **goto** en vez de en la instrucción designada por el punto de regreso original. De acuerdo con los detalles de la definición e implementación del lenguaje, en especial en cuanto a los valores finales de parámetros de valor-resultado y resultado en estas cadenas de llamada abortadas, la implementación correcta de este proceso puede ser bastante compleja.

7.3.2 Ambientes comunes explícitos

Un *ambiente común* establecido de manera explícita para compartir objetos de datos es el método más sencillo para compartir datos. A un conjunto de objetos de datos que se van a compartir entre un conjunto de subprogramas se le asigna almacenamiento en un bloque con nombre por separado. Cada subprograma contiene una declaración que nombra explícitamente el bloque compartido. Los objetos de datos dentro del bloque son entonces visibles dentro del subprograma y se puede hacer referencia a ellos por nombre en la forma usual. Un bloque compartido de este tipo se conoce por diversos nombres: *bloque COMMON* (*COMÚN*) en FORTRAN; en Ada es una forma de *paquete*; en C las variables individuales marcadas como *extern* (*externas*) se comparten de esta manera. Las *clases* de C++ y Smalltalk proporcionan esta característica, pero no es ése el propósito normal de las clases (véase el capítulo 8). El término *ambiente común* es apropiado aquí, en el presente contexto.

Especificación. Un ambiente común es idéntico a un ambiente local para un subprograma, excepto que no es parte de un subprograma individual. Puede contener definiciones de variables, constantes y tipos, pero no subprogramas o parámetros formales. La especificación de **package** (paquete) en Ada es un ejemplo:

```
package Tablas_Compartidas is
   Tamaño_Tab: constant integer := 100;
   type Tabla is array (1 . . Tamaño.Tab) of real;
   Tabla1, Tabla 2: Tabla;
   Entrada_Act:integer range 1 . . Tamaño_Tab;
end
```

La especificación de paquete define un tipo, una constante, dos tablas y una variable entera que juntos representan un grupo de objetos de datos (y definiciones de tipo) que varios subprogramas necesitan. La definición del paquete se da fuera de los subprogramas que utilizan las variables.

Si un subprograma *P* requiere acceso al ambiente común definido por este paquete, entonces se incluye un enunciado **with** explícito entre las declaraciones de *P*:

```
with Tablas_Compartidas;
```

Dentro del cuerpo de *P*, cualquier nombre contenido en el paquete se puede usar ahora directamente, como si fuera parte del ambiente local para *P*. Usaremos el nombre calificado *nombre de paquete.nombre de variable* para hacer referencia a estos nombres. Así pues, podemos escribir, en *P*:

```
Tablas_Compartidas.Tabla1(Tablas_Compartidas.Entrada_Act) :=
    Tablas_Compartidas.Tabla2(Tablas_Compartidas.Entrada_Act) + 1
```

sin más declaraciones de cualquiera de estos nombres. El nombre del paquete se debe usar como prefijo en cada nombre porque un subprograma puede usar muchos paquetes, algunos de los cuales pueden declarar el mismo nombre. (Si no hay este tipo de conflictos, el prefijo de nombre de paquete se puede evitar por medio de un enunciado **use** en Ada, por ejemplo, **with** Tablas_Compartidas **use** Tablas_Compartidas.) En general, cualquier cantidad de otros subprogramas pueden emplear el mismo ambiente común si se incluye el enunciado **with** Tablas_Compartidas, y un solo subprograma puede usar cualquier número de ambientes comunes.

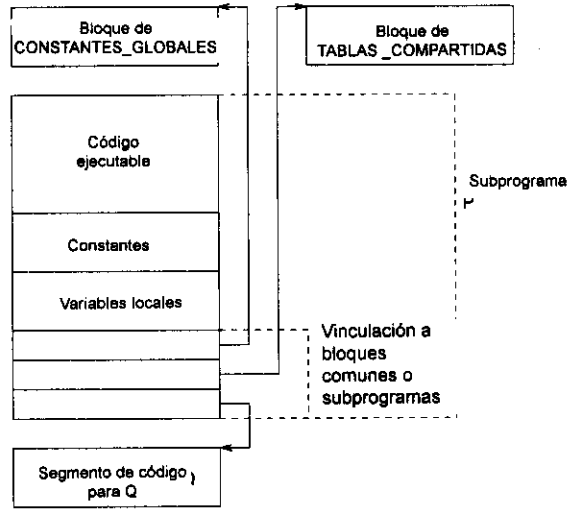
Implementación. En FORTRAN y C, cada subprograma que utiliza el ambiente común debe incluir también declaraciones para cada variable compartida para que el compilador conozca las declaraciones pertinentes, no obstante que el ambiente común también se declara en otro lugar. En Ada, se espera que el compilador recupere la declaración para el ambiente común de una biblioteca u otra parte del texto del programa cuando se encuentra un enunciado **with** durante la compilación de un subprograma.

Las declaraciones para el ambiente común se agregan a la tabla de símbolos del compilador como un conjunto adicional de nombres locales a los que se puede hacer referencia en el subprograma. Cada referencia a un nombre en el cuerpo del subprograma se consulta luego en la tabla en la forma usual para fines de verificación estática de tipos y generación de código ejecutable.

En tiempo de ejecución, un ambiente común se representa como un bloque de almacenamiento que contiene los objetos de datos declarados en la definición. Puesto que los objetos de datos son potencialmente de tipo mixto y sus declaraciones se conocen durante la compilación, el bloque se puede tratar como si fuera un *registro*. El nombre del bloque representa el nombre del registro y las variables individuales contenidas en el bloque representan los componentes del registro. Las referencias a variables individuales dentro del bloque se traducen luego al cálculo usual de dirección base más desplazamiento para referencias de componentes de registro.

El bloque de almacenamiento que representa un ambiente común debe continuar existiendo en la memoria en tanto cualquiera de los programas que lo usan sean todavía potencialmente susceptibles de llamada, porque cada activación de cualquiera de estos subprogramas puede acceder al bloque. Así pues, al bloque se asigna ordinariamente almacenamiento en forma estática de modo que se establezca al inicio de la ejecución del programa y se retenga a lo largo de la misma. Puede haber muchos bloques comunes de este tipo en la memoria, entremezclados con los bloques que representan segmentos de código para subprogramas y otros bloques de almacenamiento en tiempo de ejecución asignados estáticamente.

Un subprograma que hace referencia a un objeto de datos en un bloque común debe conocer la dirección base del bloque. Una implementación sencilla consiste en asignar una localidad en el segmento de código para que el subprograma contenga un apuntador al bloque (es decir, que contenga la dirección base del bloque). Estos vínculos de apuntador entre un subprograma y los bloques comunes que él mismo utiliza son similares a los vínculos entre un subprograma y los segmentos de código para el subprograma que él mismo llama. Una de las tareas



primordiales del editor de vínculos que ensambla una colección de subprogramas y bloques comunes en la memoria antes del inicio de la ejecución es el almacenamiento de los apuntadores de vinculación mismos en cada segmento de código según se requiere para la ejecución. La figura 7.15 ilustra esta estructura. Durante la ejecución, una referencia a un objeto de datos en un ambiente común se maneja tomando la dirección base del bloque común apropiado del segmento de código correspondiente al subprograma que hace la referencia, y sumando el desplazamiento previamente calculado para obtener la localidad efectiva en la memoria del objeto de datos compartido.

Compartimiento de variables explícitas

Una forma relacionada de compartición explícita de objetos de datos consiste en proporcionar un medio para que un objeto de datos en el ambiente *local* de un subprograma se haga visible para otros subprogramas. En esta forma, en vez de un grupo de variables en un ambiente común independiente de cualquier subprograma, cada variable tiene un "propietario", el subprograma que la declara. Para hacer visible una variable local fuera del subprograma, se debe proporcionar una *definición de exportación* explícita, como la declaración **defines** (define) en:

```
procedure P(. . .);
   defines X, Y, Z;    - X, Y y Z quedan disponibles para exportación
   X, Y, Z: real;      - Declaraciones usuales para X, Y y Z
   U, V: integer;      - Otras variables locales
begin . . . end;       - Enunciados
```

Otro subprograma que requiere acceso a una variable exportada utiliza una *definición de importación* explícita para importar la variable, por ejemplo, incluyendo una declaración **uses** (usa) que nombre tanto el subprograma como la variable exportada.

```
procedure Q(. . .);
   uses P.X, P.Z;      - Importa X y Z de P
   Y: integer;         - Otras declaraciones
begin . . . end;       - Los enunciados puede incluir referencias a X y Z
```

Éste es el modelo que presenta C para la declaración externa

Implementación. El efecto es similar al uso de una variable en un ambiente común. El almacenamiento para una variable local exportada se debe retener entre activaciones del subprograma que la define, de manera que ordinariamente se le asignaría almacenamiento en el segmento de código correspondiente al subprograma, como en el caso de las variables locales ordinarias retenidas. La referencia de una variable de esta clase desde otro subprograma que la importa utiliza entonces la dirección base del segmento de código para el subprograma exportador y suma el desplazamiento apropiado.

7.3.3 Alcance dinámico

Una alternativa al uso de ambientes comunes para datos compartidos es la asociación de un *ambiente no local* con cada subprograma en ejecución. El ambiente no local para un subprograma *P* consiste en un conjunto de ambientes locales de otras activaciones de subprograma que se hacen accesibles para *P* durante su ejecución. Cuando se hace referencia a una variable *X* en *P* y *X* carece de asociaciones locales, entonces el ambiente no local se usa para determinar la asociación correspondiente a *X*. ¿Cuál debe ser el ambiente no local para *P*? En lenguajes estructurados en bloques, las reglas de alcance estático determinan el ambiente no local implícito para cada subprograma; este enfoque más bien complejo se considera en la sección siguiente. Aquí se examina una alternativa más sencilla, aunque de uso menos extendido: el uso de los ambientes locales para subprogramas de la cadena dinámica actual.

Considérese un lenguaje en el cual los ambientes locales se eliminan a la salida del programa y donde las definiciones de subprograma no están anidadas una dentro de otra. Cada una está definida por separado respecto a las otras. Este caso, presente en APL, LISP y SNOBOL4, carece de una estructura estática de programa en la cual basar reglas de alcance para referencias a identificadores *no locales*. Por ejemplo, si un subprograma *P* contiene una referencia a *X*, y *X* no está definido localmente dentro de *P*, entonces, ¿cuál definición para *X* en algún otro subprograma se debe usar? La respuesta natural se encuentra considerando la *cadena dinámica* de activaciones de subprograma que conduce a la activación de *P*. Considérese el curso de la ejecución del programa: supóngase que el programa principal llama un subprograma *A*, el cual llama a *B*, el cual llama a *P*. Si *P* hace referencia a *X* y no existe una asociación para *X* en *P*, entonces es natural recurrir al subprograma *B* que llamó a *P* y preguntar si *B* tiene una asociación para *X*. Si *B* la tiene, entonces se emplea esa asociación; en caso contrario, se recurre al subprograma *A* que llamó a *B* y se verifica si *A* tiene una asociación para *X*. Se ha usado la *asociación de creación más reciente* para *X* en la cadena dinámica de llamadas de subprograma que conduce a *P*. Este significado para una referencia no local se conoce como *regla de la asociación más reciente*; es una regla de referencia que se basa en el alcance dinámico.

Si el ambiente no local se determina por la regla de la asociación más reciente, no se usan reglas de alcance estático; es decir, no se hace intento alguno durante la traducción del programa para determinar la definición asociada con una referencia a un identificador que no está definido localmente en el subprograma. Durante la ejecución del programa, cuando se crea un objeto de datos *X* como parte de la activación del subprograma *P*, el *alcance dinámico* de la asociación para *X* se vuelve todas las activaciones de subprograma llamadas por *P* o por esos subprogramas, y así sucesivamente. *X* es visible dentro de este alcance dinámico (excepto cuando está oculto por un subprograma posterior que tiene su propia asociación local para *X*). Visto desde el otro sentido, el *ambiente no local* de una activación de subprograma *P* se compone de la cadena dinámica completa de activaciones de subprograma que conduce a ella.

El aspecto más problemático de este ambiente no local es que puede *cambiar* entre activaciones de *P*. Así, en una activación de *P*, donde se hace una referencia no local a *X*, la asociación más reciente en la cadena de llamada puede tener a *X* como nombre de un arreglo. En una segunda activación de *P*, invocada a través de una serie distinta de llamadas de subprograma previas, la cadena dinámica puede cambiar de modo que la asociación más reciente para *X* es como el nombre de una cadena de caracteres. En una tercera activación de *P*, puede no haber en absoluto una asociación para *X* en la cadena de llamada, con lo cual la referencia a *X* es un error. Esta variabilidad en cuanto a la asociación para *X* significa que se requiere *verificación dinámica de tipos*; por tanto, este método se usa sólo en lenguajes como LISP, APL y SNOBOL4 donde la verificación dinámica de tipos se emplea por otras razones.

Implementación. La implementación de la regla de la asociación más reciente para referencias no locales es sencilla, dada la implementación de pila central para guardar registros de activación de subprograma. El ambiente local para cada subprograma se hace parte de su registro de activación. Al entrar al subprograma, se crea el registro de activación; al devolver, ese registro se elimina.

Supóngase que un subprograma *P* llama un subprograma *Q*, el cual llama a *R*. Cuando *R* se está ejecutando, la pila central podría tener el aspecto que se muestra en la figura 7.16. Para resolver una referencia no local a *X*, se hace una búsqueda en la pila comenzando con el ambiente local para *R* y retrocediendo a través de las asociaciones de la pila hasta que se encuentra la asociación más reciente para *X*. Como muestra la figura, algunas de las asociaciones de la pila están ocultas por asociaciones posteriores para el mismo identificador.

Esta implementación de la regla de la asociación más reciente es costosa. La búsqueda que se requiere en cada referencia no local toma tiempo e introduce la necesidad de guardar alguna representación de los *identificadores mismos* en las tablas de asociaciones locales, porque la posición de la asociación para *X* puede diferir en cada tabla local. Por tanto, no es posible un cómputo de dirección base más desplazamiento en la referencia no local.

¿Cómo se puede evitar la búsqueda de referencias no locales? Es posible un compromiso entre el costo de hacer referencias no locales y el costo de entrada y salida de subprogramas, el cual puede ser ventajoso si se supone que la referencia no local es mucho más frecuente que la entrada y salida de subprogramas, es decir, si es probable que el ambiente no local se use con más frecuencia que aquélla con la cual se modifica.

La implementación alternativa utiliza una tabla central común a todos los subprogramas, la *tabla central de ambientes de referencia*. La tabla central se establece de modo que contenga,

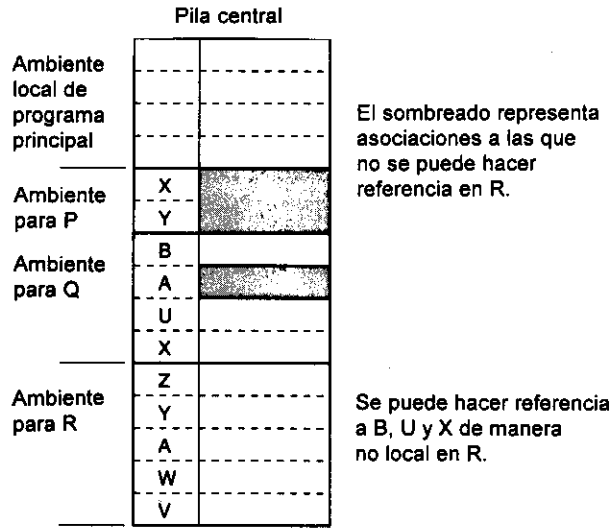


Figura 7.16. Ambiente de referencia activo durante la ejecución.

en todo momento durante la ejecución del programa, *todas las asociaciones actualmente activas de identificador*, sin que importe si son locales o no locales. Si se supone, también para simplificar, que el conjunto de identificadores a los que se hace referencia en cualquiera de los subprogramas se puede determinar durante la traducción, entonces la tabla central se inicializa de modo que contenga una entrada por cada identificador, independientemente del número de subprogramas distintos en los cuales aparece el identificador. Cada entrada de la tabla también contiene una *bandera de activación* que indica si ese identificador particular tiene una asociación activa o no, así como espacio para un apuntador al objeto de la asociación.

Todas las referencias en subprogramas se hacen directo a esta tabla central empleando el esquema de dirección base más desplazamiento ya descrito. Puesto que la asociación actual para el identificador X siempre está ubicada en el mismo lugar de la tabla central, independientemente del subprograma en el que ocurre la referencia, y sin que importe si la referencia es local o no local, este cálculo simple de referencia es adecuado. Cada referencia requiere sólo que la bandera de activación de la entrada se verifique para asegurar que la asociación de la tabla está activa actualmente. A través del uso de la tabla central se ha alcanzado el objetivo de una referencia no local relativamente eficiente sin búsqueda.

La entrada y la salida de subprograma son más costosas porque cada cambio en el ambiente de referencia requiere que se modifique la tabla central. Cuando el subprograma P llama a Q, la tabla central se debe modificar para reflejar el nuevo ambiente local para Q. Por tanto, cada entrada que corresponde a un identificador local para Q se debe modificar para incorporar la nueva asociación local para Q. Al mismo tiempo, si la antigua entrada de tabla para un identificador estaba activa, la entrada se debe guardar de modo que se pueda reactivar cuando Q sale de P. Puesto que es probable que las entradas que requieren modificación estén dispersas por toda la tabla central, esta modificación se debe hacer poco a poco, entrada por entrada. Al salir de Q, las asociaciones desactivadas y guardadas al entrar a Q se deben restaurar y reactivar.

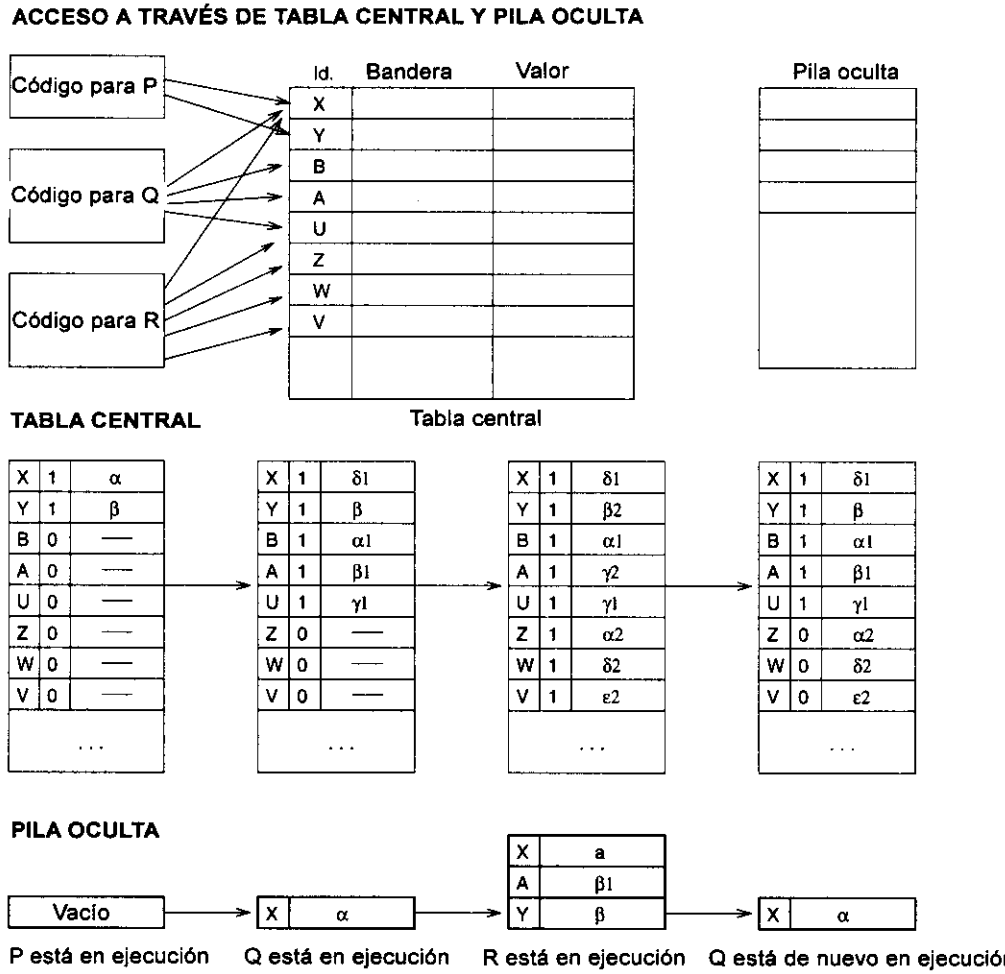


Figura 7.17. Tabla central de ambiente para referencia no local.

Una vez más, se requiere una pila en tiempo de ejecución, como en las simulaciones anteriores, pero aquí se usa como *pila oculta* para guardar las asociaciones desactivadas. Conforme cada asociación de identificador local se actualiza al entrar a Q, la antigua asociación se apila en un bloque sobre la pila oculta. Al devolver desde Q, el bloque superior de asociaciones de la pila se restaura en las posiciones apropiadas en la tabla central. Esta simulación de tabla central se muestra en la figura 7.17. Se agrega una ventaja adicional cuando se usa la tabla central si el lenguaje no permite generar referencias nuevas durante la ejecución. En este caso, como ocurría antes en relación con las tablas locales, los identificadores mismos se pueden descartar de la tabla porque nunca se van a volver a usar, pues han sido reemplazados

por el cómputo de dirección base más desplazamiento. (En cierto sentido, el identificador está representado simplemente por su desplazamiento de tabla durante la ejecución.)

7.3.4 Alcance estático y estructura de bloques

En lenguajes como Pascal y Ada, que utilizan una estructura de bloque para los programas, el manejo de referencias no locales a datos compartidos es más complejo. Si se examinan de nuevo las reglas de alcance estático para programas estructurados en bloques dadas en la sección 7.2.3, se advertirá que cada referencia a un identificador dentro de un subprograma está asociada con una definición para ese identificador en el texto del programa, incluso si el identificador no es local para el subprograma. Por tanto, el ambiente no local de referencia de cada subprograma durante la ejecución ya está determinado por las reglas de alcance estático empleadas durante la traducción. El problema de la implementación es conservar la congruencia entre las reglas de alcance estático y dinámico, de modo que una referencia no local durante la ejecución del programa se relacione correctamente con el objeto de datos que corresponde a la definición para ese identificador en el texto del programa.

La figura 7.18 muestra un ejemplo de las reglas de alcance estático para un programa estructurado en bloques en Pascal. El subprograma R es llamado desde el subprograma Q, el cual es llamado desde P. Los subprogramas P y Q y el programa principal definen una variable X. Dentro de R, se hace referencia a X de manera no local. Las reglas de alcance estático definen la referencia a X como una referencia a la X declarada en el programa principal, no como una referencia a una u otra de las definiciones para X dentro de P o Q. El significado de la referencia no local a X es independiente de la cadena dinámica particular de llamadas que conduce a la activación de R, en contraste con la regla de la asociación más reciente de la sección anterior, la cual relaciona X con la X ya sea de P o de Q, según cuál programa haya sido llamado por R.

La implementación de las reglas de alcance estático en el compilador es sencilla. Conforme cada definición de subprograma se procesa durante la compilación, se crea una tabla local de declaraciones y se anexa a una cadena de tablas locales de esta clase que representan los ambientes locales del programa principal y de otros subprogramas dentro de los cuales este subprograma está anidado. Así, pues, al compilar R el compilador agrega la tabla local de declaraciones para R a la cadena que contiene sólo la definición del programa principal. Durante la compilación, se busca en esta cadena hasta encontrar una declaración para una referencia a X, comenzando por las declaraciones locales para R y retrocediendo a lo largo de la cadena hasta las declaraciones del programa principal. Cuando se completa la compilación de R, el compilador elimina de la cadena la tabla local para R. Adviértase la similitud con la búsqueda de un significado para X hecha con la regla de la asociación más reciente descrita en la sección 7.3.3; sin embargo, esta búsqueda de una declaración para X se hace durante la *compilación*, no durante la ejecución. Las cadenas de tablas locales de declaraciones representan el anidamiento estático de definiciones de subprograma en el texto del programa, en vez de la cadena dinámica de llamado de subprogramas durante la ejecución.

En el transcurso de la ejecución del programa en un lenguaje estructurado en bloques, se utiliza una pila central para registros de activación de subprograma. El ambiente local para

```
program Main;
    var X, Y: integer;
    procedure R;
        var Y: real;
        begin
        ...
        X := X+1; /* Referencia no local a X */
        ...
        end R;
    procedure Q;
        var X: real;
        begin
        ...
        R; /* Llamar procedimiento R */
        ...
        end Q;
    procedure P;
        var X: Boolean;
        begin
        ...
        Q; /* Llamar procedimiento Q */
        ...
        end P;
begin /* comenzar Principal */
...
P; /* Llamar procedimiento P */
    ...
end.
```

Figura 7.18. Procedimiento en Pascal con referencias no locales.

cada subprograma se guarda en su registro de activación. La dificultad para mantener el alcance estático usando reglas de alcance dinámico se hace manifiesta en la figura 7.19, la cual muestra el contenido de la pila central durante la ejecución del subprograma *R* de la figura 7.18. Cuando *R* se está ejecutando y se encuentra una referencia no local a *X*, la operación de referencia debe encontrar la asociación para *X* en el programa principal, más bien que la que se halla en el subprograma *Q*, el cual llamó a *R*. Desafortunadamente, una simple búsqueda descendiente en la pila conduce a una asociación para *X* en *Q*. El problema es que la serie de tablas locales de la pila representa el *anidamiento dinámico* de *activaciones* de subprograma, el anidamiento que se basa en la cadena de llamado en tiempo de ejecución. Pero es el *anidamiento estático* de *definiciones* de subprograma el que ahora determina el ambiente no local, y la pila, tal como está estructurada actualmente, no contiene información acerca del anidamiento estático.

Para completar la implementación es necesario representar la estructura estática de bloques durante la ejecución de tal manera que se pueda usar para controlar la referencia no local. Obsérvese que en muchos aspectos la regla para referencia no local es en este caso similar a la correspondiente a referencia no local usando la regla de la asociación más reciente: para

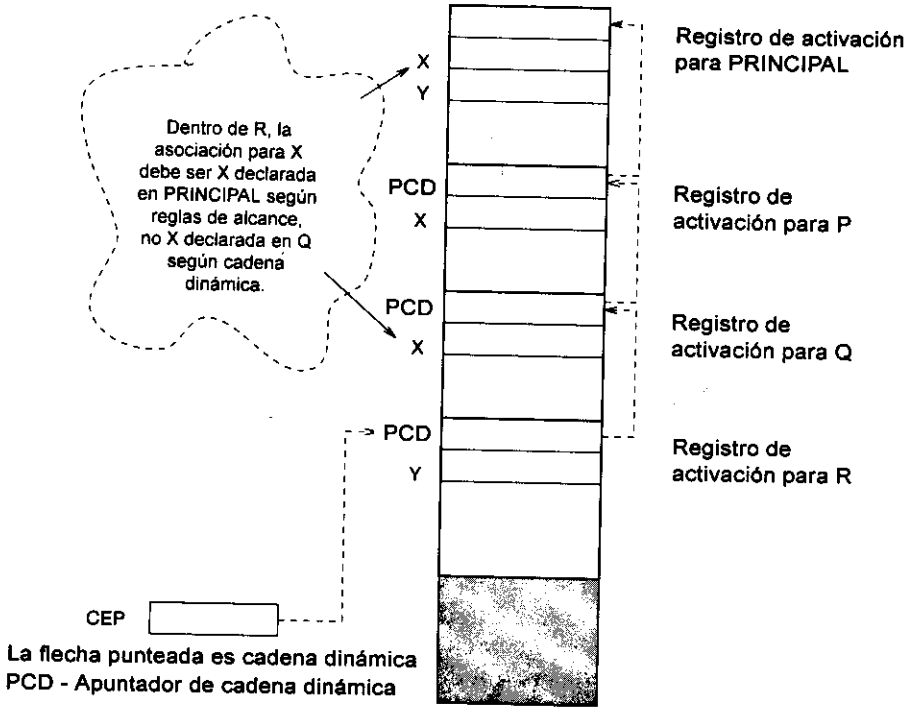


Figura 7.19. Pila central incompleta durante la ejecución usando alcance estático.

encontrar la asociación para satisfacer una referencia a *X*, se busca en una cadena de tablas de ambiente local hasta que se encuentra una asociación para *X*. Sin embargo, la cadena de tablas de ambiente local por examinar no se compone de *todas* las tablas locales que están actualmente en la pila, sino sólo de aquellas *que representan bloques o subprogramas cuya definición encierra estáticamente la definición del subprograma actual en el texto del programa original*. Por tanto, la búsqueda se hace todavía en algunas de las tablas de la pila, pero sólo en aquellas que son efectivamente parte del ambiente de referencia .

Implementación de cadenas estáticas. Estas observaciones conducen a la implementación más directa del ambiente de referencia correcto: la técnica de *cadena estática*. Supóngase que se modifican ligeramente las tablas de ambiente local en la pila de modo que cada tabla se inicie con una entrada especial, llamada *apuntador de cadena estática*. Este apuntador de cadena estática siempre contiene la dirección base de otra tabla local más abajo en la pila. La tabla señalada es la tabla que representa el ambiente local del bloque o subprograma estáticamente envolvente en el programa original. (Desde luego, puesto que cada tabla de ambiente local es simplemente una parte de un registro de activación, se puede usar la dirección base del registro de activación, en vez de sólo la dirección base de la parte de ambiente local.)

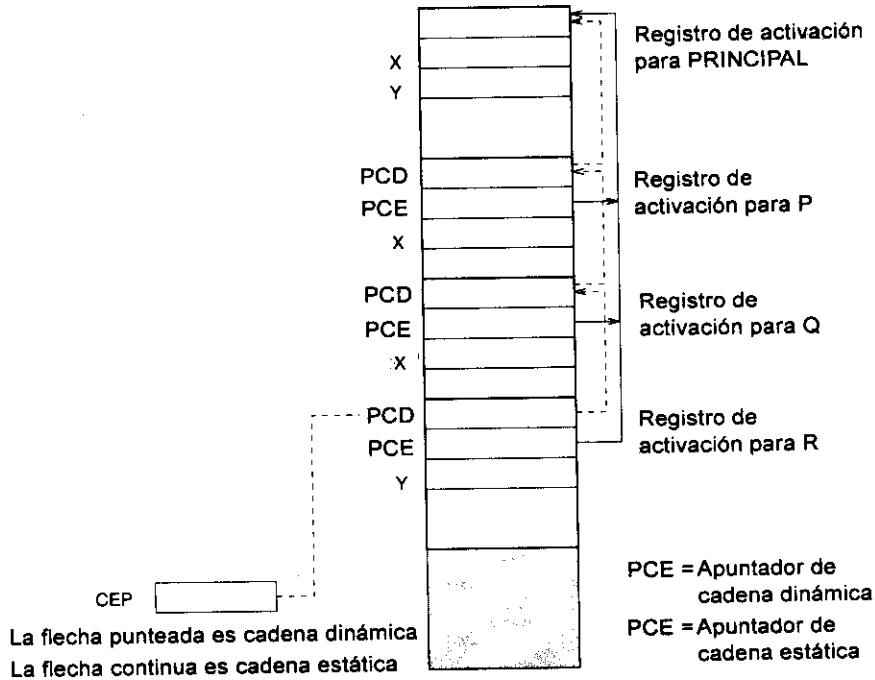


Figura 7.20. Pila central durante la ejecución.

Los apuntadores de cadena estática constituyen la base para un sencillo esquema de referencia. Para satisfacer una referencia a *X*, se sigue el apuntador CEP al ambiente local actual en el tope de la pila. Si no se encuentra una asociación para *X* en el ambiente local, entonces se sigue el apuntador de cadena estática de esa tabla local hacia abajo de la pila hasta una segunda tabla. Si *X* no está en esa tabla, la búsqueda continúa hacia abajo por los apuntadores de cadena estática hasta que se encuentra una tabla local con una asociación para *X*. La primera que se encuentra es la asociación correcta. La figura 7.20 ilustra la cadena estática para el programa en Pascal de la figura 7.18.

Si bien esto requiere en apariencia una *búsqueda* de cada ambiente estáticamente vinculado hasta que se encuentra *X*, esto no es necesariamente cierto. Conforme el compilador crea la tabla de ambiente local para cada declaración local (por ejemplo, figura 7.7), lleva la cuenta de cuáles subprogramas están anidados estáticamente dentro de ese ambiente. Para cada referencia a *X*, el compilador *cuenta* el número de ambientes envolventes en los que debe buscar (en tiempo de compilación) para encontrar la asociación correcta a *X*, y luego genera código para seguir el apuntador de cadena estática ese número de veces para llegar al registro de activación apropiado que contiene la asociación para la variable *X*. Esto evita la necesidad de guardar el nombre del identificador dentro de la pila de ejecución, como se indicó previamente.

Ésta es una implementación bastante eficiente, que en general requiere *n* accesos del apuntador de cadena estática para tener acceso a una variable declarada en el nivel *k* cuando

local de *R*, la tabla local para el bloque que lo contiene directamente, y la tabla local para el bloque más exterior (el programa principal). En las figuras 7.18 y 7.20, por ejemplo, la cadena estática para *R* siempre tiene una longitud de 2.

2. En esta cadena de longitud constante, una referencia no local siempre se satisfará en exactamente el mismo punto en la cadena. Por ejemplo, en la figura 7.20 la referencia no local a *X* en *R* siempre será satisfecha por la segunda tabla de la cadena. Una vez más, este hecho es un simple reflejo de la estructura estática del programa. El número de niveles de anidamiento estático que se deben recorrer hacia afuera de la definición de *R* para encontrar la declaración para *X* es fijo durante la compilación.

3. La posición en la cadena en la cual se va a satisfacer una referencia no local se puede determinar en tiempo de compilación. Se usó este hecho al contar el número de apuntadores de cadena estática por seguir en nuestra implementación de cadena estática anterior. Por ejemplo, se puede determinar en tiempo de compilación que una referencia a *X* en *R* se encontrará en la segunda tabla a lo largo de la cadena estática durante la ejecución. Además, se conoce en tiempo de compilación la posición relativa de *X* en esa tabla local. Así, por ejemplo, en *tiempo de compilación* se puede concluir que la asociación para *X* será la segunda entrada de la segunda tabla a lo largo de la cadena estática durante la ejecución.

Ahora está clara la base para una operación de referencia eficiente. En vez de buscar explícitamente un identificador a lo largo de la cadena estática, sólo es necesario saltar a lo largo de la cadena un número fijo de tablas y luego usar el cómputo de dirección base más desplazamiento para elegir la entrada apropiada en la tabla. Representamos un identificador en forma de una pareja (posición de cadena, desplazamiento) durante la ejecución. Por ejemplo, si *X*, a la que se hace referencia en *R*, se puede encontrar como la tercera entrada en la primera tabla a lo largo de la cadena, entonces, en el código compilado para *R*, *X* puede estar representado por la pareja (1, 3). Esta representación proporciona un algoritmo de referencia bastante simple.

En esta implementación, la cadena estática actual se copia en un vector por separado, llamado *visualizador*, al entrar a cada subprograma. El visualizador es independiente de la pila central y se suele representar en un conjunto de registros internos de alta velocidad. En cualquier punto dado durante la ejecución, el visualizador contiene la misma serie de apuntadores que se presenta en la cadena estática del subprograma que se está ejecutando actualmente. La figura 7.21 ilustra el visualizador para el ejemplo de la figura 7.18.

La referencia con uso de un visualizador es particularmente simple. Adoptemos una representación ligeramente modificada para los identificadores durante la ejecución. Una vez más, se van a usar parejas de enteros, pero en una pareja como (3,2) permítase que 3 represente el número de pasos de regreso desde el *final* de la cadena hasta el registro de activación apropiado (en vez de a partir del inicio de la cadena, como antes). El segundo entero de la pareja continúa representando el desplazamiento en el registro de activación. Ahora, dada una referencia no local como (3,10), la asociación apropiada se encuentra en dos pasos:

1. Considérese la primera entrada (3) como un subíndice al interior del visualizador. *dirección_base* = *visualizador*[3] asigna *dirección_base* como un apuntador a la dirección base del registro de activación apropiado.

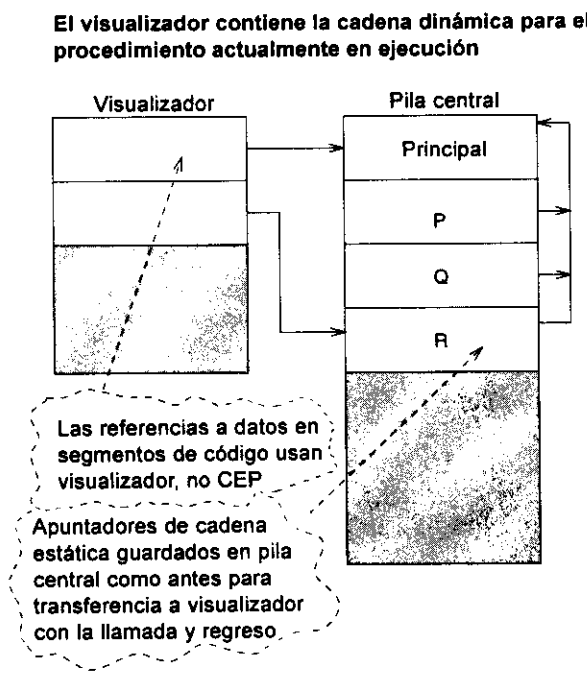


Figura 7.21. Pila central y visualizador durante la ejecución.

2. Calcúlese la localidad de la entrada deseada como dirección base más desplazamiento, como en *dirección_base* + 10.

Ordinariamente, estos dos pasos se combinan en uno usando direccionamiento indirecto a través de la entrada del visualizador. Si el visualizador se representa en registros internos de alta velocidad durante la ejecución, entonces sólo se requiere un acceso a la memoria por referencia de identificador.

Aunque las referencias se simplifican usando un visualizador, la entrada y la salida de subprograma son más difíciles porque el visualizador se debe modificar en cada entrada y salida para reflejar la cadena estática *actualmente activa*. El procedimiento más sencillo consiste en mantener los apuntadores en la pila central, como se describió anteriormente, y volver a cargar el visualizador con los apuntadores de cadena estática apropiados en cada entrada y salida, usando instrucciones insertadas por el compilador en el prólogo y el epílogo de cada segmento de código de subprograma.

Declaraciones en bloques locales

Los lenguajes como C permiten la declaración de variables locales para un bloque de enunciados, anidadas dentro de un procedimiento, como en:

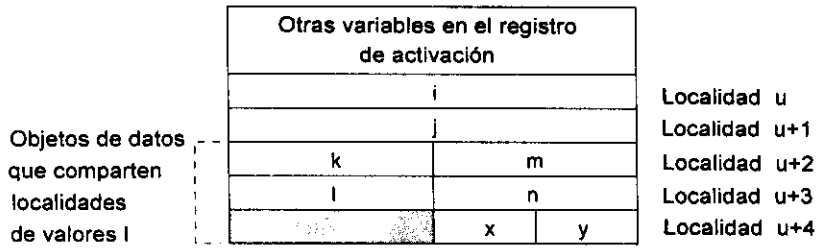


Figura 7.22. Almacenamiento de variables traslapado en el registro de activación.

```
real proc1( parámetros)
    {int i, j;
    ... /* enunciados */
    {int k, l; ... /* enunciados */ }
    {int m, n;
    ... /* enunciados */
        {int x; ... /* enunciados */ }
        {int y; ... /* enunciados */ }
        } }
```

A primera vista, parece que cada uno de los bloques requiere un registro de activación por separado para el almacenamiento de estas variables. Sin embargo, hay una diferencia importante entre estos bloques y el análisis previo de los registros de activación de procedimiento. Debido a la naturaleza dinámica de las llamadas de procedimiento, nunca se puede estar seguro (en general) de cuáles procedimientos están actualmente en ejecución. De modo que, si todos los bloques anteriores representan procedimientos, podría ser el caso que el bloque que contiene a *k* y el bloque que contiene a *m* estén ambos en ejecución de manera simultánea (por ejemplo, el bloque que contiene a *k* llama al bloque que contiene a *m*), por lo cual necesitarían registros de activación por separado.

Sin embargo, en el ejemplo anterior en C esto es imposible. Dentro del bloque que contiene la variable *k* no se puede estar dentro del alcance de la variable *m*. De manera similar, si se está en el alcance de *x* no se puede estar dentro del alcance de *y*. Esto permite una estrategia de almacenamiento sencilla, que es análoga a la estructura de almacenamiento para registros variables de la figura 4.13. Las variables *k* y *l* pueden usar las mismas localidades de almacenamiento que *m* y *n*, puesto que no pueden estar activas al mismo tiempo, y todo este almacenamiento se asigna con el procedimiento de encierro. Todo lo que hacen las declaraciones locales es definir la visibilidad, el conjunto de enunciados, donde se puede usar cada variable. El almacenamiento existe para el procedimiento completo de activación. Esto se sintetiza en la figura 7.22.

7.4 LECTURAS ADICIONALES SUGERIDAS

Casi todos los libros y artículos a los que se hace referencia en el capítulo 1 y que se ocupan de las estructuras de control a nivel de subprogramas también tratan los problemas asociados

de ambientes de referencia, datos compartidos y parámetros. Los textos sobre escritura de traductores que se mencionan en el capítulo 3 describen cómo compilar esta clase de estructuras.

7.5 PROBLEMAS

1. Cuando los ambientes locales de referencia se eliminan entre activaciones de subprogramas a través del uso de una pila central, como en Pascal, a veces parece como si los valores se *conservaran*. Por ejemplo, en casi todas las implementaciones de Pascal, si el procedimiento *Sub* tiene una variable local *X* y *Sub* asigna el valor 5 a *X* en la primera llamada, entonces en una segunda llamada, si se hace referencia (involuntariamente) a *X* antes de que se le asigne un nuevo valor, a veces *X* todavía tiene su antiguo valor de 5. Sin embargo, en el mismo programa una tercera llamada a *Sub* puede encontrase con que *X no* conserva su antiguo valor de la segunda llamada.
   (a) Explique esta aparente anomalía: ¿en qué circunstancias una activación de *Sub* que hace referencia a una variable no inicializada *X* podría encontrar que *X* todavía tiene el valor asignado en una llamada anterior? ¿En qué circunstancias *X* no tendría el valor previamente asignado?
   (b) Escriba un programa simple en Pascal y determine los efectos de intentar el proceso anterior en su implementación local.

2. Suponga que un lenguaje permite especificar valores iniciales para variables locales, por ejemplo, la declaración en Ada:
   X: integer := 50
   la cual inicializa *X* a 50. Explique los dos significados que una inicialización como esta podría tener en los casos donde *(a)* las variables locales se *conservan* entre llamadas y *(b)* las variables locales se *eliminan* entre llamadas.

3. El Pascal (y muchos otros lenguajes) permiten usar variables de tipo *apuntador* junto con una operación *new* (nuevo) para construir objetos de datos. Pascal también emplea el enfoque de eliminación en ambientes locales de referencia. La combinación tiene un alto potencial de generación de *basura* (almacenamiento inaccesible) durante la ejecución del programa. Explique la dificultad.

4. Para un programa que usted haya escrito recientemente en un lenguaje que utiliza *reglas de alcance estático*, tome cada subprograma y enumere los nombres que hay en *(a)* su ambiente local de referencia, *(b)* su ambiente no local de referencia, *(c)* su ambiente global de referencia, y *(d)* su ambiente de referencia predefinido. Luego invierta el problema: para un nombre declarado en cada uno de estos ambientes de referencia, enumere *(a)* las definiciones de subprograma en su alcance estático y *(b)* las activaciones de subprograma en su alcance dinámico.

5. El objetivo del *ocultamiento de información* en un diseño de subprograma se suele dificultar si el lenguaje no permite retener datos locales entre llamadas (como en Pascal). Proporcione un ejemplo de una situación en Pascal donde la eliminación de ambientes

locales hace necesario volver visible un objeto de datos fuera de un subprograma, no obstante que el subprograma es la única rutina que debería tener acceso al objeto de datos.

6. En la implementación de *visualizador* del referencia no local en los lenguajes estructurados en bloques, ¿se puede predecir el tamaño *máximo* del visualizador durante la compilación de un programa? ¿Cómo?

7. En lenguajes como Pascal, que emplean reglas de alcance estático y permiten llamadas recursivas de subprogramas, a veces es difícil relacionar una referencia no local en un subprograma con una variable particular en otro subprograma si existen varias activaciones recursivas de ese segundo subprograma. Por ejemplo, suponga que *A*, *B*, *C* y *D* son los subprogramas del programa *Principal*, y *A* es llamado por *Principal*. Suponga que *A* llama a *B*, *B* llama a *A* de manera recursiva, *A* llama a *C* y luego *C* llama a *D*.
(a) Si *D* hace una referencia no local a la variable *X* de *A*, entonces, ¿cual "X" es visible en *D*, la de la primera activación de *A* o la de la segunda?
(b) Con los patrones de llamada y referencia no local que aquí se suponen, sólo son posibles *cuatro* anidamientos estáticos distintos de las definiciones de *A*, *B*, *C* y *D*. Dibuje los cuatro anidamientos estáticos posibles (por ejemplo, como en la figura 7.5).
(c) Para cada uno de los cuatro anidamientos estáticos, dibuje la pila en tiempo de ejecución en el momento en que se hace referencia a "X" en *D*, mostrando los registros de activación para *A*, *B*, *C* y *D* y las cadenas estática y dinámica. Para cada caso, proporcione la representación en tiempo de ejecución de "X" como una pareja (n,m) y explique cuál objeto de datos *X* debe recuperar la referencia no local a "X" en *D* dada esta estructura en tiempo de ejecución. Suponga la implementación de apuntador de cadena estática de la referencia no local.

8. Suponga que desea modificar el Pascal de modo que la referencia no local se base en alcance dinámico (asociación más reciente) en vez de alcance estático.
(a) ¿Cómo se tendría que modificar la organización de la pila central en tiempo de ejecución? ¿Qué información se podría eliminar de cada registro de activación de subprograma? ¿Qué información se tendría que agregar a cada registro de activación?
(b) Explique cómo se podría resolver una referencia no local en esta implementación modificada?
(c) Proporcione dos ejemplos de revisiones en busca de errores que se tendrían que hacer en tiempo de ejecución (esto se puede hacer en tiempo de compilación si se usa alcance estático).

9. Suponga que usted desea proyectar un lenguaje que utilice *(1)* ambientes locales retenidos, *(2)* nada de llamadas recursivas de subprogramas, y *(3)* referencias no locales con base en alcance dinámico. Proyecte una implementación para un lenguaje como ése. Explique cómo se representa el ambiente de referencia. Explique las acciones que se adoptan al llamar y devolver un subprograma. Explique cómo se implementó la referencia no local.

10. Explique por qué es *imposible*, en un lenguaje con sólo parámetros transmitidos por valor o nombre, como ALGOL, escribir un subprograma *Intercambio* de dos parámetros que simplemente intercambia los valores de sus dos parámetros (los cuales deben ser variables simples o subindizadas). Por ejemplo, *Intercambio* llamado por *Intercambio*(*X*,*Y*)

debería devolver con *X* con el valor original de *Y* y *Y* con el valor original de *X*. Suponga que *Intercambio* funciona sólo para argumentos de tipo entero.

11. Considere el programa siguiente:

```
program Principal(...);
   var Y: integer;
   procedure P(X: integer);
      begin X := X + 1; write (X,Y) end;
   begin
   Y := 1; P(Y); write(Y)
   end.
```

Proporcione los tres números que se imprimen en caso de que *Y* se transmita a *P (a)* por valor, *(b)* por referencia, *(c)* por valor-resultado, y *(d)* por nombre.

12. Considere el programa siguiente, parecido a Pascal:

```
program principal(input,output);
   var i,j,k,m: integer;
   procedure Q(var i: integer; m: integer);
      begin
      i:=i+k;
      m:=j+1;
      writeln(i,j,k,m)
      end;
   procedure P(var i: integer, j: integer);
      var k: integer;
      begin
      k:=4;
      i:=i+k;
      j:=j+k;
      Q(i,j)
      end;
begin
i:=1;
j:=2;
k:=3;
P(i,k);
writeln(i,j,k)
end.
```

Complete la tabla siguiente para cada enunciado *writeln* suponiendo los mecanismos de paso de parámetros dados:

Modo de parámetros	i	j	k	m
Pascal, como se escribe				
Todos los parámetros son de llamada por referencia				
Todos los parámetros son de llamada por valor				
Todos los parámetros son de llamada por valor-resultado				

13. Las referencias a arreglos usando subíndices computados, por ejemplo, $A[I]$, se consideran a veces como una forma de *creación de seudónimos*. Por ejemplo, los dos nombres compuestos $A[I]$ y $A(J]$ se pueden considerar como seudónimos. Explique por qué estas referencias a arreglos crean algunos de los mismos problemas para el programador y el implementador que otros tipos de seudónimos.

14. El FORTRAN requiere que la declaración de cada objeto de datos que se comparte a través de un ambiente común se proporcione de nuevo en cada subprograma que utiliza el objeto de datos. Esto permite compilar cada definición de subprograma de manera independiente, puesto que cada una contiene declaraciones completas para objetos de datos compartidos. Suponga que el compilador proporciona involuntariamente una declaración ligeramente distinta en cada uno de los subprogramas *Sub*1 y *Sub*2 que comparten una variable *X*. ¿Que ocurre si *Sub*1 y *Sub*2 se compilan en forma independiente y luego se vinculan y se ejecutan? Suponga que el objeto de datos real *X* se representa en tiempo de ejecución de acuerdo con la declaración dada en *Sub*1 y no con la de *Sub*2.

15. Suponga que el subprograma de la figura 7.10 es llamado desde dentro de *P* usando *constantes* como parámetros reales, por ejemplo, $Q(2,3)$ en vez de $Q(a,\&b)$. Aunque la transmisión de una constante por referencia no es legal en C, se permite en muchos lenguajes, de modo que suponga que es válida en este caso. Recuerde que (1) a un objeto de datos constante se le asigna almacenamiento ordinariamente en el segmento de código del subprograma de llamada, y (2) una expresión de parámetro real que consiste en una sola constante es un caso especial del hecho más general de una expresión arbitraria que aparece como parámetro real. Proporcione dos métodos por los cuales se podría implementar la transmisión de una constante por valor y por referencia. El método 1 no usa almacenamiento temporal en el que llama; el método 2 sí lo utiliza. (El método 1 usado para transmisión por referencia puede conducir a "constantes" que cambian de valor; por ejemplo, en ciertas implementaciones de FORTRAN se puede escribir un programa que suma 1 y 2 y obtiene 4. Escriba un programa así y pruébelo en su implementación local. [*Sugerencia:* Pruebe con el equivalente de:

    Sub(1); X := 2; X := X + 1; **print**(X)

y escriba *Sub* de modo que modifique la literal 1 para que tenga el valor 2.)

16. *Dispositivo de Jensen*. Los parámetros transmitidos por nombre permiten el uso de un truco de programación que se conoce como *dispositivo de Jensen*. La idea básica es transmitir por nombre, como parámetros por separado a un subprograma, tanto una expresión en la que interviene una variable o más como las variables mismas. Mediante cambios hábiles en los valores de las variables acopladas con referencias al parámetro formal que

corresponde a la expresión, la misma se puede evaluar para muchos valores distintos de las variables. Un ejemplo sencillo de la técnica se encuentra en la rutina de sumatoria para usos generales *Suma*, definida en ALGOL como sigue:

```
real procedure Suma (Expr, Índice, LI, LS);value LI, LS;
   real Expr; integer Índice, LI, LS;
   begin real Temp; Temp := 0
   for Índice := LI step 1 until LS do Temp:=Temp+Expr;
   Suma := Temp
   end Suma;
```

En este programa *Expr* e *Índice* se transmiten por nombre, y LI y LS por valor. La llamada de:

```
Suma(A[I],I,1,25)
```

devolverá la suma de los primeros 25 elementos del vector *A*. La llamada de:

```
Suma(A[I] * B[I],I,1,25)
```

devolverá la suma de los productos de los primeros 25 elementos correspondientes de los vectores *A* y *B* (si se supone que *A* y *B* han sido declarados en forma apropiada). La llamada:

```
Suma(C[K,2],K,-100,100)
```

devolverá la suma de la segunda columna de la matriz *C* desde *C*[−100,2] hasta *C*[100,2].

(a) ¿Qué llamada a *Suma* daría la suma de los elementos que están sobre la diagonal principal de una matriz *D*, declarada como **real array** D[1:50, 1:50]?

(b) ¿Qué llamada a *Suma* daría la suma de los cuadrados de los primeros 100 números impares?

(c) Utilice el dispositivo de Jensen para escribir una rutina *Máx* que regresa el valor máximo de un conjunto de valores obtenido evaluando una expresión arbitraria *Expr* que contiene un índice *Índice*, el cual varía en un intervalo de *LI* a *LS* en pasos de tamaño *Paso* (un entero).

(d) Muestre cómo se puede obtener el efecto del dispositivo de Jensen usando subprogramas como parámetros en un lenguaje sin transmisión de parámetros por nombre.

Capítulo 8

Abstracción II: Herencia

En la sección 5.1 se estudió el concepto de tipos de datos encapsulados como un medio para proyectar programas que crean nuevos tipos de datos con operaciones que trabajan sólo sobre objetos de estos tipos nuevos. Por ejemplo, para organizar las clases, se podrían definir los tipos *sección* y *estudiante* y se podrían implementar operaciones como *AgregarASección* con la signatura *estudiante* × *sección* → *sección* para manejar la suma de un nuevo estudiante a una sección de cursos dada.

Dado el programa:

```
typedef { definición } sección;
typedef { definición } estudiante;
sección ClaseNueva;
estudiante EstudianteNuevo;
...
AgregarASección(EstudianteNuevo,ClaseNueva);
```

los detalles de cómo se implementan objetos *estudiante* y *sección* están ocultos por el subprograma *AgregarASección*. El programador puede considerar *estudiante* y *sección* como tipos primitivos y *AgregarASección* como una función primitiva. Sólo el implementador del subprograma *AgregarASección* necesita estar consciente de la estructura real de los diversos tipos de datos.

Aunque este tipo de técnicas se pueden emplear en casi todos los lenguajes, sería preferible hacer su uso más fácil y menos propenso a errores. En vez de confiar en que los programadores "hagan lo correcto", nos gustaría que el lenguaje nos ayudara a encapsular datos.

En primer término se describen mecanismos para encapsular datos de manera automática, como el **package** (paquete) de Ada. Luego se amplía este concepto de manera que se puedan obtener en forma automática operaciones sobre estos objetos de datos usando un concepto que se llama *herencia*. Tales operaciones se reconocen como *métodos*. Por último, se amplía el concepto al polimorfismo pleno en operaciones.

8.1 UNA NUEVA VISITA A LOS TIPOS DE DATOS ABSTRACTOS

Recuérdese, de la sección 5.1, que se define un *tipo de datos abstracto* como un nuevo tipo de datos definido por el programador y que incluye:

1. Un tipo de datos definido por el programador,
2. Un conjunto de operaciones abstractas sobre objetos de ese tipo, y
3. Encapsulamiento de objetos de ese tipo de tal manera que el usuario del nuevo tipo no pueda manipular esos objetos excepto a través del uso de las operaciones definidas.

La *abstracción de datos*, es decir, el diseño de objetos de datos abstractos y de operaciones sobre esos objetos, es una parte fundamental de la programación, como ya se ha expuesto. En un lenguaje de programación que suministra poco apoyo directo para abstracción de datos más allá del mecanismo ordinario de subprogramas, el programador puede proyectar y usar de todos modos sus propios tipos de datos abstractos, pero el concepto no está presente en el lenguaje mismo. En su lugar, el programador debe utilizar "convenciones de codificación" para organizar su programa de modo que se consiga el efecto de un tipo de datos abstracto. Sin embargo, sin el apoyo por parte del lenguaje para la definición de tipos de datos abstractos, el encapsulamiento de un tipo nuevo no es posible. Por tanto, si se violan las convenciones de codificación, ya sea de manera intencional o no, la implementación del lenguaje no puede detectar la violación. Estos tipos de datos abstractos creados por el programador suelen aparecer como bibliotecas especiales de subprogramas en lenguajes como C, FORTRAN y Pascal. Ada y C++ se cuentan entre los pocos lenguajes de uso extendido con características de abstracción de datos.

Las definiciones de tipos, como las que suministra C, simplifican la declaración de nuevas variables del tipo, puesto que sólo se necesita el nombre del tipo en la declaración. Sin embargo, la estructura interna de los objetos de datos del tipo no está encapsulada. A cualquier subprograma que pueda declarar una variable como del nuevo tipo también se le permite tener acceso a cualquier componente de la representación del tipo. Por tanto, cualquier subprograma de esta clase puede pasar por alto las operaciones definidas sobre los objetos de datos y en su lugar tener acceso directo a los componentes de los objetos de datos y manipularlos. La intención del encapsulamiento de una definición de tipo de datos abstracto es imposibilitar ese acceso para que los únicos subprogramas que sepan cómo están representados los objetos de datos del tipo sean las operaciones definidas como parte del tipo mismo.

De los lenguajes que se describen en la parte II, sólo Ada, C++ y Smalltalk proporcionan apoyo de lenguaje para encapsulamiento de estas definiciones de tipos abstractos. En Ada, una definición de tipo abstracto de éstas es una forma de un *paquete*. Un paquete que define un tipo de datos abstracto *TipoSección* puede adoptar la forma que se muestra en la figura 8.1. La declaración **is private** (es privado) para el tipo *Sección* indica que la estructura interna de los objetos de datos de sección no va a ser accesible desde subprogramas que utilizan el paquete. Los detalles mismos de este tipo se proporcionan al final del paquete en el componente **private** del paquete. Sólo los subprogramas que están dentro de la definición del paquete tienen acceso a estos datos privados. Así, por ejemplo, los procedimientos *AsignarEstudiante* y *CrearSección* pueden tener acceso al arreglo *ListaDeClase* que es un componente de una

```
package TipoSección is
   type IDEstudiante is integer;
   type Sección(TamañoMáx: Integer) is private;
   procedure AsignarEstudiante(Secc: in out Sección;
      Estud in IDEstudiante;
   procedure CrearSección(Secc: in out Sección;
      Prof in integer;
      Salón in integer);
   private
   type Sección(TamañoMáx: Integer) is
      record
         Salón: integer;
         Profesor: integer;
         TamañoDeClase: integer range 0..TamañoMáx :=0;
         ListaDeClase: array (1..TamañoMáx) of IDEstudiante;
      end record;
   end;
package body TipoSección is
   procedure AsignarEstudiante(...) is
         - Enunciados para insertar estudiante en ListaDeClase
      end;
   procedure CrearSección(...) is
         - Enunciados para inicializar componentes de registro de Sección
      end;
   procedure ProgramarSalón(...) is
         - Enunciados para programar Sección en un salón
      end;
   end;
```

Figura 8.1. Tipo de datos abstracto Sección definido en Ada.

Sección, pero cualquier otro procedimiento (fuera del paquete) no puede acceder a este componente, aunque el otro procedimiento pueda declarar que una variable es del tipo *Sección* (y especificar un tamaño máximo de clase como parámetro).

Implementación. En los tipos de datos abstractos definidos como paquetes en Ada intervienen pocas ideas nuevas de implementación. Un paquete proporciona encapsulamiento para un conjunto de definiciones de tipo y subprogramas. Así pues, su nuevo efecto primordial es restringir la visibilidad de los nombres declarados en el paquete, de manera que los usuarios del tipo abstracto no puedan tener acceso a los elementos internos de la definición. Una vez que el compilador determina que un procedimiento dado tiene acceso a la definición de tipo, los algoritmos de capítulos anteriores para asignación y acceso a objetos de datos son aplicables.

Después de haber señalado que los paquetes de Ada no emplean nuevas ideas de implementación, es interesante examinar más de cerca el concepto de **package** (paquete). Cada paquete de Ada contiene dos partes: una parte de especificación y una parte de implementación. Como está dada en la figura 8.1, la especificación para el paquete *TipoSección* define todos

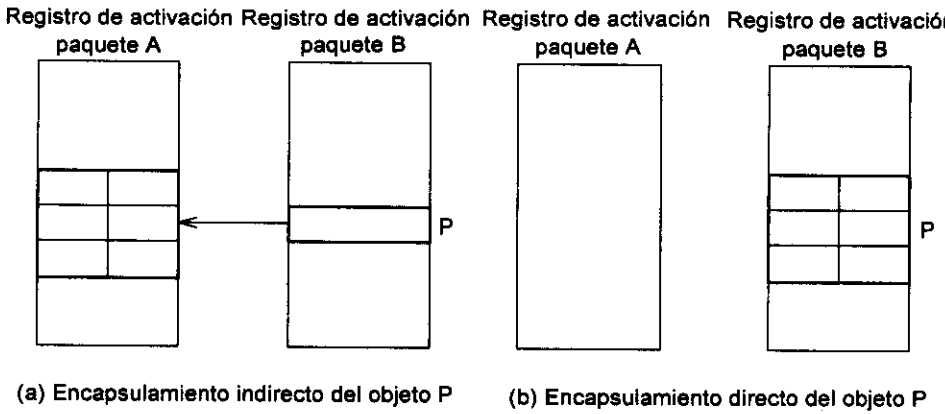


Figura 8.2. Dos modelos de implementación para datos abstractos.

los datos, tipos y subprogramas que son conocidos y hechos visibles para procedimientos declarados en otros paquetes. La implementación de los procedimientos declarados en la parte de especificación del paquete están dados en el componente **package body** (cuerpo de paquete). El cuerpo de paquete puede incluir también objetos de datos y tipos adicionales que no van a hacerse visibles para otros paquetes. En este caso, el procedimiento *ProgramarSalón* sólo se puede llamar desde dentro de *AsignarEstudiante* y *CrearSección*; el nombre simplemente no es conocido fuera del paquete.

La necesidad de incluir el nombre del procedimiento en la especificación del paquete es bastante simple: el compilador necesita conocer la signatura del procedimiento si éste es llamado desde otro paquete, y sólo la especificación del paquete define los tipos para cada uno de los parámetros formales. Así pues, una llamada a *AsignarEstudiante* desde un procedimiento de otro paquete necesita la información de que la llamada requiere dos parámetros reales, uno un argumento **in out** de tipo *Sección* y el otro un parámetro **in** del tipo *IDEstudiante*.

Sin embargo, ¿por qué se colocan las definiciones de datos privados en la especificación del paquete, si los mismos son información que se conoce y utiliza sólo dentro del cuerpo de paquete? Parecería más sencillo permitir que esta clase de declaraciones se colocaran sólo dentro del componente cuerpo de paquete. Para entender por qué Ada ha definido los paquetes de esta manera, es necesario examinar las dos implementaciones típicas para tipos de datos abstractos.

La figura 8.2 presenta dos modelos para implementar objetos de datos encapsulados. La figura 8.2(a) es un ejemplo de *encapsulamiento directo*. En este caso, la estructura del tipo de datos abstracto está definida por la especificación de paquete *A*. El almacenamiento mismo para el objeto *P* se mantiene en un registro de activación para el paquete *A*. En el paquete *B*, que declara y utiliza el objeto *P*, el registro de activación en tiempo de ejecución debe contener un apuntador al almacenamiento de datos real.

En la figura 8.2(b) se presenta una implementación alternativa, llamada *encapsulamiento directo*. Como en el caso indirecto, la estructura del objeto de datos abstracto está definida

```
package A is
   type MiPila is private;
   procedure PilaNueva(S: out MiPila);
   ...
   private
   type MiPilaRep;
      — Detalles ocultos de MiPila
   type MiPila is access MiPilaRep;
   — B sólo tiene apuntador a pila
end;
```

(a) Encapsulamiento indirecto

```
package A is
   type MiPila is private;
   procedure PilaNueva(S: out MiPila);
   ...
   private
type MiPila is record
   Top: integer;
   A: array (1..100) of integer;
   end record;
      — B tiene estructura de pila
end;
```

(b) Encapsulamiento directo

Figura 8.3. Dos ejemplos de encapsulamiento para datos de Ada.

por la especificación para el paquete *A*. Sin embargo, en este caso el almacenamiento real para el objeto *P* se mantiene dentro del registro de activación para el paquete *B*.

¿A qué se debe la diferencia? En el caso indirecto, la implementación del tipo de datos abstracto es auténticamente independiente de su uso. Si la estructura de *P* cambia, sólo el paquete *A* necesita cambiar. El paquete *B* únicamente necesita saber que el objeto *P* es un apuntador y no requiere conocer el formato de los datos a los que *P* señala. Para sistemas grandes con miles de módulos, el tiempo que se ahorra al no volver a compilar cada módulo cada vez que la definición de *P* cambia es significativo.

Por otra parte, el acceso a *P* tiene un costo en tiempo de ejecución. Cada acceso a *P* requiere un acceso indirecto de apuntador para llegar al almacenamiento mismo. Mientras que un solo uso no es significativo, el acceso repetido a *P* puede ser costoso.

El caso del encapsulamiento directo tiene las características opuestas. En este caso, el objeto de datos *P* se guarda dentro del registro de activación del paquete *B*. El acceso a los componentes de *P* puede ser más rápido, puesto que se puede usar el acceso a datos normalizados de base más desplazamiento en un registro de activación local; no es necesaria una indirección a través de un apuntador. Sin embargo, si la representación del objeto abstracto cambia, entonces todos los casos de su uso (por ejemplo, el paquete *B*) también se deben volver a compilar. Esto hace que los cambios de sistema sean costosos en cuanto a tiempo de compilación, pero más eficientes en cuanto a ejecución.

Ada utiliza el modelo de encapsulamiento directo para contribuir a una eficiencia de ejecución máxima en tiempo de proceso. La traducción del uso de un objeto de datos abstracto (por ejemplo, el paquete *B* de la figura 8.2) requiere detalles de la representación del objeto, de ahí la necesidad de la sección **private** en la especificación del paquete.

Adviértase, sin embargo, que se puede usar el encapsulamiento directo o el indirecto en cualquier programa que maneje encapsulamiento, independientemente del modelo que se implemente como parte de la arquitectura de software del lenguaje de programación. Aunque el encapsulamiento directo es el modelo preferido e implícito dentro de Ada, el encapsulamiento indirecto es posible si lo implementa así el programador. La figura 8.3 presenta segmentos en Ada que demuestran ambas estrategias de implementación. La figura 8.3(a)

```
package TipoPilaEnt is
   type Pila(Tamaño: Positive) is private;
   procedure Push(I: in integer; S: in out Pila);
   procedure Pop(I: out integer; S: in out Pila);
private
   type Pila(Tamaño: Positive) is record
      AlmacenamientoPila; array (1 .. Tamaño) of integer;
      Top: Integer range 0 .. Tamaño := 0;
      end record;
end TipoPilaEnt;
package body TipoPilaEnt is
   procedure Push(I: in integer; S: in out Pila) is
      begin
         — Procedimiento de cuerpo de Push
      end;
   procedure Pop(I: out integer; S: in out Pila) is
      begin
         — Procedimiento de cuerpo de Pop
      end;
end TipoPilaEnt;
```

Figura 8.4. Abstracción de pila de enteros en Ada.

corresponde a la implementación de la figura 8.2(a), y la figura 8.3(b) corresponde a la implementación de la figura 8.2(b). (**access** es una variable apuntador en Ada.)

Una ligera variante del encapsulamiento directo de la figura 8.3(b) está dada por la definición de tipo:

```
package A is
   type MiPila is record
      Top: integer;
      A: array (1..100) of integer;
      end record;
   ...
```

En este caso, la organización del registro de activación es igual que el encapsulamiento directo; sin embargo, todos los nombres son visibles dentro del paquete *B*. Éste es el mecanismo usual en lenguajes como Pascal, los cuales suministran tipos de datos sin encapsulamiento.

Tipos de datos abstractos genéricos

Los tipos primitivos de datos integrados en un lenguaje suelen permitir al programador declarar el tipo básico de una clase nueva de objetos de datos y luego especificar también varios atributos de los objetos de datos. Esta es una forma sencilla de polimorfismo que se analizará con más detalle en la sección 8.3. Por ejemplo, Pascal suministra el tipo básico de datos

```
generic
   type Elem is private;
package TipoCualquierPila is
   type Pila(Tamaño: Positive) is private;
   procedure Push(I: in Elem; S: in out Pila);
   procedure Pop(I: out Elem; S: in out Pila);
private
   type Pila(Tamaño: Positive) is record
      AlmacenamientoPila: array (1 .. Tamaño) of Elem;
      Top: Integer range 0 .. Tamaño := 0;
      end record;
end TipoCualquierPila;
package body TipoCualquierPila is
   procedure Push(I: in Elem; S: in out Pila) is
      begin
         — Procedimiento de cuerpo de Push
      end;
   procedure Pop(I: out Elem; S: in out Pila) is
      begin
         — Procedimiento de cuerpo de Pop
      end;
end TipoCualquierPila;
```

Figura 8.5. Abstracción de pila genérica en Ada.

arreglo sobre el cual están definidas varias operaciones primitivas, como la subindización. Sin embargo, la definición de tipo:

```
type Vect = array [1..10] of real;
```

especifica atributos adicionales de la clase de objetos de datos de tipo *Vect*: cada uno tiene 10 componentes de tipo real, accesibles a través de los subíndices 1, 2, ... , 10. Se pueden escribir subprogramas adicionales para manipular objetos de datos *Vect*, pero las operaciones que suministra el tipo base *arreglo* también continúan disponibles. Es deseable una estructura similar para la definición de tipos de datos abstractos. Por ejemplo, para definir un nuevo tipo abstracto *pila* con operaciones de *agregar* y *remover* para insertar y eliminar elementos en la pila, la pila se podría definir como un paquete de Ada igual que en la figura 8.4). Adviértase el problema que se presenta: el tipo de elemento en un objeto de datos de pila es parte de la definición de tipo para *pila*, de modo que esta definición es para un tipo de datos *pila de enteros*. Un tipo *pila de reales* o *pila de secciones* requiere una definición de paquete por separado, no obstante que la representación de la pila y las operaciones de *agregar* y *remover* se pueden definir de manera idéntica.

Una *definición genérica de tipo abstracto* permite especificar por separado un atributo del tipo de esta clase, de modo que se pueda dar una definición de tipo base, con los atributos como parámetros, y luego se puedan crear varios tipos especializados derivados del mismo tipo base. La estructura es similar a la de una definición de tipo con parámetros, excepto que

en este caso los parámetros pueden afectar la definición de las operaciones en la definición del tipo abstracto, así como las definiciones mismas de tipo, y los parámetros pueden ser nombres de tipo y también valores. El paquete de Ada de la figura 8.5 muestra una definición genérica de tipo de esta clase para un tipo *pila genérica* en el cual tanto el tipo de elemento guardado en la pila como el tamaño máximo de la misma están definidos como parámetros.

Ejemplarización de una definición genérica de tipo abstracto. Una definición genérica de paquete representa una *plantilla* que se puede usar para crear tipos de datos abstractos particulares. El proceso de crear la definición particular del tipo a partir de la definición genérica para un conjunto dado de parámetros se llama *ejemplarización*. Por ejemplo, dada la definición genérica del tipo pila en Ada de la figura 8.5, se puede ejemplarizar para producir una definición del tipo *pilaent* equivalente al de la figura 8.4 por medio de la declaración:

```
package TipoPilaEnt is
   new TipoCualquierPila(elem => integer);
```

Un tipo de datos pila que contiene "secciones" se puede definir por medio de la ejemplarización:

```
package TipoPilaFijo is
   new CualquierTipoPila(elem => Sección);
```

Posteriormente se pueden declarar pilas de enteros de distintos tamaños:

```
Pila1: TipoPilaEnt.Pila(100);
PilaNueva:TipoPilaEnt.Pila(20);
```

y en forma similar, se pueden declarar pilas de secciones:

```
PilaSec: TipoPilaFijo.Pila(10);
```

Adviértase que el tipo genérico *CualquierTipoPila* se puede ejemplarizar muchas veces para muchos valores distintos de los parámetros, y cada ejemplarización produce otra definición para el nombre de tipo *Pila* dentro del paquete. Por tanto, cuando se hace referencia a pila en una declaración, ello puede ser ambiguo. Ada requiere que el nombre del paquete anteceda al nombre del tipo en la declaración para resolver la ambigüedad, por ejemplo, *TipoPilaEnt.pila* o *TipoPilaFijo.Pila*.

En C++, un concepto similar se conoce como *template* (*plantilla*) y se puede usar para definir clases genéricas:

template <class *nombre_de_tipo*> **class** *nombredeclase definición_de_clase*

lo cual define un conjunto infinito de definiciones de clase, para todos los argumentos *nombre_de_tipo*.

Implementación. En principio, un tipo de datos abstracto genérico tiene por lo común una implementación sencilla. Los parámetros para el paquete genérico deben estar dados en el programa cuando la definición del paquete se ejemplariza. El compilador utiliza la definición genérica del paquete como una plantilla, inserta los valores especificados para los parámetros y luego compila la definición lo mismo que si fuera una definición ordinaria de paquete sin parámetros. Esto es comparable a la capacidad de la macro *#define* de C. Durante la ejecución del programa, sólo aparecen objetos de datos y subprogramas; la definición del paquete sirve principalmente como un dispositivo para restringir la visibilidad de estos objetos de datos y subprogramas. El paquete mismo no aparece como parte de la estructura en tiempo de ejecución.

Si una definición genérica de tipo se ejemplariza muchas veces (como podría ocurrir, por ejemplo, si el paquete genérico se suministrara en una biblioteca), entonces esta implementación sencilla puede ser demasiado ineficiente, porque la ejemplarización produce una copia del paquete completo, incluidos todos los subprogramas que se definen en el paquete, los cuales se deben volver a compilar entonces por completo. Una mejor implementación evitaría la generación de una nueva copia de cada subprograma y también evitaría la recompilación total del paquete completo. Esto se analiza más a fondo en la sección 8.3 al estudiar el polimorfismo con mayor detalle.

8.2 HERENCIA

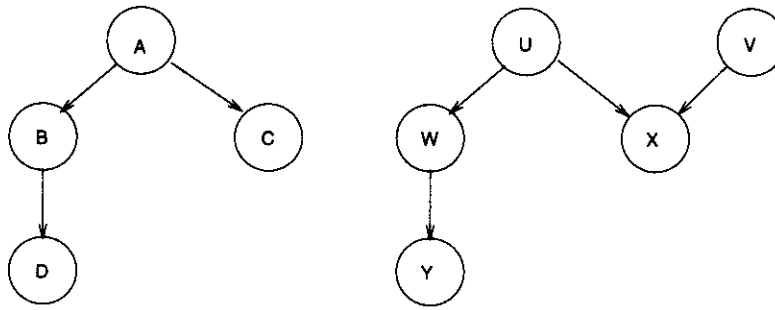
Suele ocurrir que la información que se conoce en una parte de un programa se necesita y se utiliza en otra parte. Por ejemplo, los parámetros reales de un subprograma de llamada, cuando se hace referencia a ellos como parámetros formales de un subprograma llamado, constituyen un mecanismo que pasa los valores de datos de los parámetros reales al subprograma llamado. En este caso, esta vinculación es explícita. Existe una invocación concreta de subprograma que efectúa esta asociación.

Sin embargo, con frecuencia la información se pasa entre componentes de programa de manera implícita. Al traslado de esta información le llamamos *herencia*. La herencia es la recepción, en un componente de programa, de propiedades o características de otro componente de acuerdo con una relación especial que existe entre los dos componentes. A menudo se ha usado la herencia en el diseño de lenguajes de programación.

Una forma temprana de herencia se encuentra en las reglas de alcance para datos estructurados en bloques. Los nombres que se usan en un bloque interno pueden heredarse del bloque exterior. Por ejemplo, considérese:

```
{int i, j;
    {float j, k;
    k = i + j; }
}
```

En $k = i + j$, tanto j como k son variables locales *float* declaradas en el bloque actual; sin embargo, i se hereda del bloque exterior, y la declaración de *int j* está impedida de ser heredada al bloque interior debido a la redefinición de j como una variable *float*.



(a) Herencia sencilla (b) Herencia múltiple por X

Figura 8.6. Herencia sencilla y múltiple.

Aunque las reglas de alcance en lenguajes como Ada, C y Pascal son una forma de herencia, el término se usa más a menudo para referirse al traslado de datos y funciones entre módulos independientes de un programa. La **class** (clase) de C++ es un ejemplo de este tipo. Si se establece una relación entre la clase *A* y la clase *B*, escrita $A \Rightarrow B$, entonces ciertos objetos dentro de la clase *A* se heredarán de manera implícita y se podrán usar dentro de la clase *B*. Si un cierto objeto *X* se declara dentro de la clase *A* y no se redefine en la clase *B*, entonces cualquier referencia al objeto *X* dentro de la clase *B* se refiere en realidad al objeto *X* de la clase *A* por vía de herencia, en forma muy parecida a la referencia a la variable *i* en el ejemplo anterior en C.

Si se tiene $A \Rightarrow B$, se dice que *A* es la clase *progenitora* o *superclase*, y *B* es la clase *dependiente*, *niña* o *subclase*. *A* es el *ancestro* inmediato de *B*. En la figura 8.6(a), *A* es el ancestro de *B* y de *C*, *B* y *C* son hermanas y son los descendientes inmediatos de *A*, en tanto que *D* es un descendiente de *A*. Si una clase sólo puede tener un único progenitor [figura 8.6(a)], se dice que tiene *herencia sencilla*. Si una clase puede tener múltiples progenitores [figura 8.6(b)], se dice que tiene *herencia múltiple*.

8.2.1 Clases derivadas

En lenguajes como C++, las clases están estrechamente ligadas al concepto de encapsulamiento. Las clases tienen típicamente una parte que es heredada por otra clase y una parte que se usa internamente y está oculta respecto al exterior. Por ejemplo, las pilas enteras de Ada de la figura 8.4 se pueden representar en C++ como:

```
class pilaent
    {private:
    int tamaño;
    int almacenamiento(100); }
```

El nombre *pilaent* es un nombre de clase conocido fuera de la clase misma, en tanto que las declaraciones *tamaño* y *almacenamiento* que siguen a **private**, como en Ada, representan componentes de la clase *pilaent* que sólo se conocen dentro de la clase.

Una abstracción incluye tanto descriptores de datos como funciones que pueden operar sobre objetos de ese tipo. Estas funciones se conocen con frecuencia como *métodos*. Una descripción más completa de la clase *pilaent*, incluidos sus métodos, sería:

```
class pilaent
   {public:
   pilaent() { tamaño=0;}
   void push(int i) { tamaño=tamaño+1; almacenamiento[tamaño]=i;}
   int pop ...
private:
   int tamaño;
   int almacenamiento(100);
   }
```

public se refiere a nombres que son visibles fuera de la definición de clase y pueden ser heredados por otras clases. La función *pilaent*(), de igual nombre que la clase misma, en la definición de clase es una función especial llamada *constructor*, que es llamada siempre que se crean objetos de esa clase. Aquí, esta función inicializa la pila en ceros, pero, en general, puede ejecutar cualquier código de inicialización pertinente necesario para establecer objetos de esa clase. En este ejemplo, *push* (agregar) y *pop* (remover) se pueden usar fuera de la definición de clase y se invocan como métodos aplicados a objetos de la clase *pilaent*, como en *b.push*(7) o *j* = *b.pop*() para *pilaent b*. Los nombres *tamaño* y *almacenamiento* se conocen sólo dentro de la definición de clase.

Si la función ~ *pilaent* estuviera definida, entonces sería un *destructor* y se le llamaría siempre que se fueran a desasignar objetos de esa clase. En el ejemplo anterior, la función destructor no se requiere, pero si el constructor adjudicó cierto almacenamiento adicional, el destructor se necesitaría para deshacer esa asignación y eliminar el almacenamiento adicional.

La notación *b.push*(7) no es tan extraña como parece a primera vista. Las definiciones de clase son similares a las definiciones de tipo en C, sólo que se tienen tipos ampliados con componentes de llamada de función. Por consiguiente, *b.push* es el componente de función *push* del objeto *b*. La similitud entre las clases de C++ y el **struct** de C no es simple coincidencia. La declaración en C:

```
struct A {int B}
```

es simplemente una forma taquigráfica de la declaración en C++:

```
class A {public: int B}
```

Sin embargo, las definiciones de clase pueden incluir definiciones de función y también declaraciones de datos.

```
class elem {
public:
    elem() { v=0;}
    void AElem(int b) { v = b;}
    int DesdeElem() { return v; }
private:
    int v;}

class PilaElem: elem {
public:
    PilaElem() { tamaño=0;}
    void push(elem i)
        {tamaño=tamaño+1; almacenamiento[tamaño]=i;}
    elem pop()
        {tamaño=tamaño-1; return almacenamiento[tamaño+1]}
private:
    int tamaño;
    elem almacenamiento[100]; }

{ elem x;
    PilaElem y;
    int i;
    read(i);              — Obtener valor entero para i
    x.AElem(i);           — Coerción a tipo elem
    y.push(x);            — Poner x en pila y
    ...
}
```

Figura 8.7. Clases derivadas en C++.

La herencia en C++ tiene lugar a través del uso de clases *derivadas*. Por ejemplo, si una clase crea objetos de tipo *elem*, y otra clase crea pilas de objetos de ese tipo, esto puede ser especificado por la figura 8.7. El nombre *PilaElem* en seguida de la designación **class** es la clase derivada, y el nombre que sigue a la coma es la clase base (*elem*). Si se consideran las operaciones de una clase simplemente como componentes de la definición del objeto, entonces la sintaxis se vuelve muy fácil de entender. La notación *x.AElem*(...) se convierte en una referencia al método *AElem* definido en la clase *elem*, la cual da por resultado una llamada al procedimiento *AElem* usando el objeto *x* como un argumento implícito.

Este ejemplo incluye varios aspectos importantes del uso de clases para encapsular datos:

1. Todos los nombres públicos dentro de la clase *elem* son heredados por *PilaElem*. Estos nombres también son públicos en *PilaElem* y conocidos para cualquier usuario de una pila como ésta.

2. El contenido de un objeto de tipo *elem* son datos privados no conocidos fuera de la definición de clase. En este caso, *AElem* y *DesdeElem* funcionan como operadores de coerción que convierten de tipo entero a tipo *elem*. Éstos siempre serán necesarios si la estructura interna de la clase es privada.

Las palabras clave **private** y **public** controlan la visibilidad de los objetos heredados. C++ también incluye la palabra clave **protected** (protegido), que significa que el nombre es visible para cualquier tipo derivado del tipo base, pero no se le conoce fuera de la jerarquía de la definición de clase.

Implementación. La implementación de clases no impone mucho trabajo adicional al traductor. En una clase derivada, sólo los nombres heredados de la clase base se agregan al espacio de nombre local para la clase derivada, y sólo las públicas se hacen visibles para los usuarios de esa clase. La representación misma de almacenamiento para los objetos definidos se puede determinar de manera estática a través de las declaraciones de datos dentro de la definición de clase.

Si en la definición está presente una constructora, entonces el traductor debe incluir una llamada a esa función siempre que se encuentra una declaración nueva; por ejemplo, al entrar a un bloque. Por último, para la invocación de métodos, como *x.push(i)*, el traductor sólo necesita considerar esto como una llamada a *push(x, i)* considerando *x* como un primer argumento implícito. La gestión de almacenamiento es igual que en el C estándar, y para C++ se puede usar la misma organización con base en pilas de C.

Cada ocurrencia de clase tiene su propio almacenamiento de datos consistente en los objetos de datos que constituyen objetos de la clase, así como apuntadores a todos los métodos definidos para objetos de esa clase. Si el objeto se deriva de alguna otra clase base, entonces el traductor maneja esa herencia y el almacenamiento mismo para el objeto contiene todos los detalles de su implementación. A esto se le conoce como un *enfoque con base en copia* de la herencia, y es el modelo más simple y directo de implementar.

Un modelo alternativo de implementación se conoce como *enfoque con base en delegación*. En este modelo, cualquier objeto de una clase derivada utiliza el almacenamiento de datos de la clase base. Las propiedades heredadas no se reproducen en el objeto derivado. Este modelo requiere una forma de compartición de datos para que los cambios al objeto base puedan causar cambios en el objeto derivado.

El C++ sólo utiliza el enfoque con base en copia, pero la compartición basada en la delegación conduce a un uso más eficiente de la memoria (por ejemplo, compartición de propiedades heredadas) y a la capacidad de que los cambios a un objeto se propaguen en forma automática (e instantánea) a través de la jerarquía de clases derivadas.

8.2.2 Métodos

El término *programación orientada a objetos* ha sido motivo de tanto bombo publicitario en años recientes, que el término está perdiendo rápidamente su significado. Para muchos, el término se ha vuelto sinónimo del concepto de encapsulamiento que hemos descrito anteriormente. Sin embargo, orientación a objetos significa más que la simple combinación de datos y subprogramas en un solo módulo. La herencia de métodos para crear objetos nuevos proporciona un poder adicional que va más allá del simple encapsulamiento.

```
class elem {
public:
   elem() { v=0;}
   void AElem(int b) { v = b;}
   int DesdeElem() { return v; }
private:
   int v;}

class PilaElem; elem {
public:
   PilaElem() { tamaño=0;}
   void push(elem i)
      {tamaño=tamaño+1; almacenamiento[tamaño]=i;}
   elem pop()
      {tamaño=tamaño–1; return almacenamiento[tamaño+1]}
   void MiTipo() {printf("Soy de tipo PilaElem \n")}
protected:
   int tamaño;
   elem almacenamiento[100]; }

class PilaNueva: PilaElem {
public:
   int atisbar() {return almacenamiento[tamaño].DesdeElem()} }
```

Figura 8.8. Herencia de métodos.

Por ejemplo, considérese una extensión al ejemplo de *PilaElem* dado anteriormente. En el ejemplo ampliado (figura 8.8) se agregó el procedimiento público *MiTipo* en la clase *PilaElem*, el cual imprime el nombre del tipo: "Soy de tipo PilaElem." También se hizo **protected** (protegida) la estructura interna de *PilaElem* para que se pueda heredar a cualquier clase derivada de esta clase.

En nuestra clase pila, sólo consideramos las operaciones de *push* (agregar) y *pop* (remover), las cuales anexan y eliminan, respectivamente, un elemento en la pila. Supóngase que se desea una nueva clase *PilaNueva*, la cual se comporta como una *PilaElem*, pero incluye el nuevo método *atisbar*, el cual devuelve el valor en el tope de la pila sin modificar la misma. A través de la definición de la figura 8.8, en apariencia se hace esto. *PilaNueva* hereda todas las propiedades de *PilaElem* (todos los datos y métodos públicos) y agrega el nuevo método *atisbar*. Sin saber siquiera de las operaciones *push* y *pop* de la clase *PilaElem* o *AElem* y *DesdeElem* de la clase *elem*, estas funciones quedan a disposición de cualquier objeto declarado como *PilaNueva*. Si estas clases se conservan como módulos separados mantenidos por programadores distintos, los cambios a las especificaciones de *elem* o *PilaElem*, y en forma correspondiente a las implementaciones de estas clases, serán transparentes para la definición de *PilaNueva* y operarán sobre objetos *PilaNueva* al igual que con objetos *PilaElem*.

Se tiene un problema, sin embargo. Para objetos de la clase *PilaNueva*, el método *MiTipo* continúa imprimiendo "Soy de tipo PilaElem", puesto que ésa es la definición del método heredado de la clase *tipoelem*. Esto se puede arreglar de dos maneras:

1. Se podría simplemente redefinir el método *MiTipo* en la definición de la clase *PilaNueva*:

   ```
   void MiTipo() {printf("Soy de tipo PilaNueva\n")}
   ```

   Aunque esto funcionaría, tiene la complejidad de exigir una descripción completa de cada método que requiera cambios en una clase derivada.

2. Se podría usar una función **virtual**. En una definición de método, cada nombre de subprograma que es llamado en la definición se enlaza al subprograma al cual se refiere *en el momento de definición del método*. Éste es el enlace sintáctico normal presente en lenguajes como C, Pascal, Ada, FORTRAN y en casi todos los lenguajes compilados. Por otra parte, los subprogramas **virtual** se enlazan dinámicamente *en el momento de llamada del subprograma*.

Para ver la diferencia, dentro de la definición de clase de *PilaElem*, se podría definir *MiTipo* como sigue:

```
virtual void NombreDeTipo() {printf("PilaElem\n")};
void MiTipo() {printf("Soy de tipo", NombreDeTipo())}
```

y en la clase *PilaNueva* se podría definir *NombreDeTipo* como sigue:

```
virtual void NombreDeTipo() {printf("PilaNueva\n")};
```

Aunque las diferencias entre los dos enfoques no son grandes en este ejemplo, el enlace retardado de la llamada del método virtual da cabida a cambios dinámicos en el comportamiento de ejecución de las clases. Esto sería más importante en las clases donde métodos tales como *MiTipo* fueran extremadamente largos y complejos. En vez de reproducir este método, con modificaciones, en cada clase derivada, sólo es necesario redefinir una pequeña función virtual de cambios al método en cada clase derivada.

Implementación. Los métodos virtuales se pueden implementar de manera similar a la tabla central de entorno de registros de activación (sección 7.3.3). Cada método virtual en una clase derivada reserva un segmento en el registro que define la clase. El procedimiento constructor simplemente llena la localidad del nuevo procedimiento virtual, si lo hay. En caso contrario, llena la localidad del procedimiento virtual desde la clase base.

8.2.3 Clases abstractas

Hay ocasiones en las cuales es deseable que la definición de clase sea simplemente una plantilla para una clase y no permitir que se declaren objetos con esa definición. Se presentan dos alternativas a este modelo: superclases abstractas y herencia *mixin*.

Superclase abstracta. Considérese la clase *PilaElem* con la función virtual *NombreDeTipo* antes dada. Tal como se declaró, no habría una prohibición para que un usuario incluyera:

```
PilaElem X;
```

en un programa y creara una ocurrencia de la clase *PilaElem*. Sin embargo, se podría desear que *PilaElem* fuera simplemente una plantilla de superclase y requiriera que todos los objetos que usan esta clase fueran de una subclase derivada. En C++ esto se expresa declarando *NombreDeTipo* como una función virtual nula:

```
virtual void NombreDeTipo()=0;
```

Ningún objeto se puede crear a partir de clases que tienen funciones virtuales nulas. Cualquier clase derivada debe redefinir esta función para crear ocurrencias de la clase.

Herencia mixin. El modelo de herencia que se ha presentado hasta aquí ha sido $A \Rightarrow B$, donde la clase *B* se deriva de la clase *A* y es una modificación de ella. Existe otra forma de herencia llamada herencia *mixin* (*de incorporación*) donde sólo se define la diferencia entre la clase base y la nueva clase derivada. Considérese la clase *PilaElem* antes examinada y la clase derivada *PilaNueva*. En vez de definir una nueva clase para *PilaNueva*, simplemente se define una clase *delta* que proporciona los cambios. Aunque esta opción no existe en C++, se la podría representar usando una notación similar a la de C++:

```
deltaclass ModPila
    {int atisbar() {devolver almacenamiento[tamaño].DesdeElem();}
    }
```

Se crearía entonces la nueva clase como sigue:

```
class PilaNueva = class PilaElem + deltaclass ModPila
```

la cual tendría la semántica de creación de una nueva clase *PilaNueva*, la cual hereda todas las propiedades de la clase *PilaElem* modificadas por la clase delta *ModPila*.

La ventaja respecto a la herencia mixin es que las clases delta se pueden aplicar a cualquier clase. Así, si se tuviera una clase comparable *ColaElem*, se crearía una nueva clase que atisbara el final de la cola usando la misma clase delta que se empleó para las pilas:

```
class colanueva = class ColaElem + deltaclass ModPila
```

Esta clase delta se podría aplicar en numerosas situaciones sin necesidad de redefiniciones complejas de todos los objetos de clase.

8.2.4 Objetos y mensajes

Smalltalk representa un enfoque alternativo hacia el desarrollo de objetos y métodos, el cual difiere de manera significativa en cuanto a diseño respecto a los modelos de herencia presentados previamente en este capítulo para Ada y C++. Smalltalk fue concebido por Alan Kay en el Centro de Investigación de Palo Alto de Xerox durante los primeros años de la década de 1970, aunque muchos contribuyeron al desarrollo final del lenguaje. Smalltalk fue proyectado como un entorno personal y total de cómputo. Como tal, incluía un lenguaje para representar algoritmos, así como un ambiente de cómputo compuesto de una pantalla con ventanas y un ratón en ejecución en la estación de trabajo Xerox Alto. Todo esto es bastante común en la actualidad, pero se consideraba revolucionario cuando se desarrolló por primera vez.

Un programa en Smalltalk se compone de un conjunto de definiciones de clase integradas por objetos de datos y métodos. Todos los datos están encapsulados, puesto que sólo los métodos de una clase dada tienen acceso a los datos dentro de esa definición de clase. El ocultamiento de información y el encapsulamiento son características integradas de manera inherente y no se injertaron en la estructura de tipos del lenguaje, como ocurrió con C++.

Un programa en Smalltalk se compone de tres características primordiales del lenguaje:

Definiciones de clase. Se trata de enunciados ejecutables que definen la estructura interna y los métodos que se pueden usar para crear y manipular objetos de una clase. Se pueden definir datos que son comunes para todos los objetos de una clase dada.

Ejemplarización de objetos. Se crean objetos específicos para cada definición de clase invocando métodos de creación dentro de la definición de clase. Los métodos se pueden definir para ocurrencias de una clase.

Paso de mensajes. Los métodos se pasan como mensajes a un objeto para llevar a cabo una acción. En vez de asociar un conjunto de parámetros a una función, como es el caso en casi todos los demás lenguajes, en Smalltalk se asocia una función (es decir, un método) con un objeto de datos. Esta asociación de un método con un objeto se conoce como *mensaje*.

Existen tres tipos de mensajes en Smalltalk:

1. Un mensaje *unario* es simplemente un método que no tiene parámetros. Por ejemplo, se puede invocar el método predefinido *new* (nuevo) para crear un objeto de casi cualquier clase:

```
x _ Conjunto new
```

crea un nuevo objeto de clase *Conjunto* y lo asigna a la variable *x*. *x* es ahora una ocurrencia de un objeto de la clase *Conjunto*.

2. Un mensaje *binario* se usa principalmente para operadores aritméticos. 3 + 6 se refiere a que el método binario + se está enviando al objeto 3 con el argumento 6. El método + devuelve el objeto 9 (lo que no es sorprendente) para este mensaje.

3. Los mensajes de *palabra clave* tienen un comportamiento similar al de las funciones de homonimia en lenguajes como Ada y C++. Para asignar un valor al tercer elemento del arreglo *z*, la secuencia sería crear el arreglo y luego llevar a cabo la asignación:

```
x _ Arreglo new:10
x at:3 put:42
```

El método *new*: se envía primero a la clase *Arreglo* con el parámetro 10 para crear un arreglo de 10 elementos que se asigna a la variable *x*. El método de palabras clave *at:* (en:) y *put:* (poner:) se invoca para asignar el tercer componente de *x* con el valor 42. El nombre del método para esta asignación se da como la concatenación de las palabras clave, o método *at:put:*. Otro método *at:*, si se envía a una ocurrencia de un arreglo, recupera un valor del arreglo.

Se requiere un poco de la historia de las computadoras para entender el uso del símbolo _ como operador de asignación. Originalmente, el operador de asignación en Smalltalk era el símbolo ←, el cual, durante los años sesenta y setenta, estaba en la misma posición del teclado que _, por lo cual tenía la misma definición interna. El uso de ← en el teclado ha desaparecido en términos generales, pero su posición de carácter para uso en Smalltalk no ha cambiado. En forma similar, el operador para indicar un valor devuelto desde un método era originalmente ↑, el cual comparte la posición de teclado con ∧. El uso de ↑ en los teclados también ha desaparecido en términos generales.

Las series de enunciados se pueden ejecutar en Smalltalk como un bloque:

$$[\ :variable\_local \mid enunciado_1 ... enunciado_n)$$

donde *variable_local* es una variable local optativa declarada en el bloque. (el símbolo : inicial impide ambigüedad sintáctica en las definiciones de bloque.) La ejecución de un bloque tiene lugar pasando el método *value* (valor) al bloque, y el resultado de la ejecución es la última expresión del bloque:

```
| x |
x _ ["Esto es una cadena"]. "a x se ha asignado el bloque."
x value print ! "El bloque asignado a x está evaluado."
```

lo cual provoca que:

1. Se declare la variable local *x*;
2. A *x* se le asigne el bloque;
3. *x value* origina que se devuelva *Esto es una cadena*, lo cual imprime *print*.

Los comentarios se indican mediante comillas dobles en Smalltalk. El símbolo ! (al que se suele hacer referencia como *bang* en el medio editorial en inglés y que se ha adoptado así en la jerga de computadoras) es un mandato para que Smalltalk ejecute la serie anterior de enunciados.

El uso del paso de parámetros por palabras clave permite la creación de muchas estructuras de control comunes. El ambiente predefinido de Smalltalk incluye para ocurrencias booleanas el método *ifTrue:ifFalse*: (siCierto:siFalso:). Cada una de estas palabras clave toma un bloque como argumento. Por ejemplo, *true ifTrue:ifFalse*: evalúa el bloque *ifTrue:*, mientras que *false ifTrue:ifFalse*: evalúa el bloque *ifFalse:*. Si se escribe esto usando las sangrías familiares de tipo Pascal, se obtiene la sintaxis siguiente:

```
x > 2
    ifTrue:['x es mayor que 2' printNl]
    ifFalse: ['x es menor o igual que 2' printNl]
```

En este caso, el método > con el parámetro 2 se pasa al objeto *x*. El método > devuelve *cierto* o *falso*, según corresponda. A este objeto booleano se le pasa luego el método de palabras claves *ifTrue:ifFalse*: con los dos bloques como argumentos y, según cuál ocurrencia booleana se evalúe, se ejecuta el bloque *ifTrue*: o *ifFalse*:. (*PrintNl* es similar al método *print* con la adición de un carácter de renglón nuevo impreso después de que se imprime el objeto.) La ejecución se comporta en forma muy parecida a un enunciado if-then-else, pero la ejecución misma es radicalmente distinta. Se pueden desarrollar construcciones de iteración de manera similar usando este enfoque.

Herencia de clases

Los datos en Smalltalk se basan en una jerarquía de clases. Si cualquier método que se pasa a un objeto no está definido dentro de esa clase, se pasa a la clase progenitora, y así sucesivamente. La clase *Objeto* es la superclase progenitora de todas las clases. De ese modo, la herencia de métodos es una característica primitiva del lenguaje. Smalltalk sólo maneja herencia sencilla con una sola clase ancestro, aunque los métodos se pueden pasar a través de múltiples niveles de clases progenitoras.

En los métodos de palabras clave, el parámetro para el método se nombra en forma explícita en la declaración del método, como en:

```
ifTrue: Bloquecierto ifFalse: Bloquefalso
    "Las variables Bloquecierto y Bloquefalso son parámetros
    en el cuerpo del método."
```

Sin embargo, ¿cómo se puede tener acceso al objeto al cual se pasa el método? Por ejemplo, en $x > 2$, ¿cómo tiene acceso el método > al objeto *x* para obtener su valor? Éste es el propósito del objeto *self* (mismo). Se comporta como el parámetro *this* en C++.

La necesidad y el uso del objeto *self* con jerarquías de objetos se demuestra mediante el sencillo ejemplo siguiente. Supóngase que se desean dos clases: *ClaseA* y *ClaseB*, donde *ClaseB* es una subclase de la *ClaseA*. Se pueden definir estas clases como sigue:

```
Object subclass: #ClaseA
    instanceVariableNames: ' '
    classVariableNames: ' '
    poolDictionaries: ' '
    category: nil!
ClaseA subclass: #ClaseB
    instanceVariableNames: ' '
    classVariableNames: ' '
    poolDictionaries: ' '
    category: nil!
```

ClaseA se define como una subclase de la clase *Object* (Objeto) (simplemente para colocarla en algún lugar en la jerarquía de Smalltalk), y *ClaseB* es una subclase de *ClaseA*. *instanceVariableNames* definirá el conjunto de nombres locales que se usan en cada ejemplarización de un objeto de la clase, en tanto que *classVariableNames* serán datos globales en todas las ocurrencias de objetos de clase. *poolDictionaries* y *category* no se necesitan aquí y se explicarán más tarde en la sección 12.3. Sin embargo, la sintaxis de Smalltalk requiere que se proporcionen todos los argumentos para métodos de palabras clave, incluso con argumentos nulos.

Considérense los métodos *imprímelo* y *pruébalo*, los cuales se agregan como métodos a la *ClaseA* a través del mandato *methodsFor* (métodosPara):

```
!ClaseA methodsFor: 'básico'!
    "Agregar métodos a la clase ClaseA"
imprímelo
    'Esto es clase A' printNl !
pruébalo
    self imprímelo !!
```

Cada método se define dando su nombre (y parámetros, en su caso), seguido de su definición, seguida de ! para concluir la definición. !! termina las declaraciones *methodsFor* para *ClaseA*.

La aplicación de estos métodos se puede mostrar con:

```
| x | x _ ClaseA new. x pruébalo !
```

lo cual imprime, como es de esperar:

```
Esto es clase A
```

La ejecución tiene lugar conforme:

1. *new* se pasa a la clase *ClaseA*, la cual lo pasa a la clase progenitora *Object*.
2. *Object* ejemplariza el objeto y lo asigna a la variable *x*.
3. El método *pruébalo* se pasa al objeto ejemplarizado de *ClaseA* llamado *x*.
4. El método *pruébalo* se define como *self imprímelo*, donde *self* se refiere al objeto *x*.
5. El método *imprímelo* se pasa al objeto *x*, el cual imprime la respuesta apropiada.

Sin embargo, considérese lo que ocurre cuando se define un método similar para *ClaseB*, la cual tiene un método *imprímelo* pero ningún método *pruébalo*:

```
!ClaseB methodsFor: 'básico'!
imprímelo
    'Esto es clase B' printNl !!
```

Si se escribe:

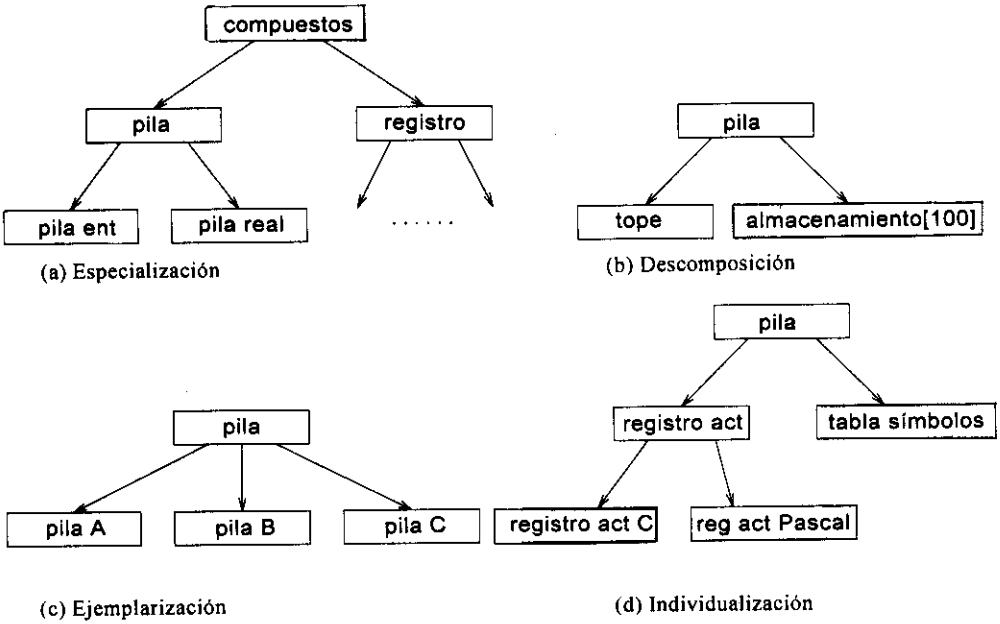


Figura 8.9. Conceptos de abstracción.

```
| x | x _ ClaseB new. x pruébalo !
```

se obtiene ahora:

```
Esto es clase B
```

El método *pruébalo* se pasa a la ocurrencia *x* de la clase *ClaseB*. Puesto que el método *pruébalo* no está definido para la clase *ClaseB*, *pruébalo* se pasa a la clase progenitora *ClaseA*. Aquí, *pruébalo* está definido. *self* todavía se refiere al objeto *x*, pero ahora es de clase *ClaseB*, de manera que se invoca el *imprimelo* de *ClaseB* en vez del método progenitor de la *ClaseA* que contiene el método *pruébalo*.

8.2.5 Conceptos de abstracción

Después de analizar el papel que desempeñan los métodos y las funciones virtuales, quizá resulte útil repasar el papel de la abstracción en los lenguajes de programación. El encapsulamiento se suele ver como un mecanismo para "dividir y vencer" con el fin de proporcionar control intelectual sobre el programa en desarrollo. Al programador sólo se le proporciona acceso a los objetos de datos que son parte de la especificación del segmento del programa por desarrollar. La especificación de cualquier otra parte del programa (y la implementación interna de datos que satisface esas especificaciones) está fuera del dominio del conocimiento del programador. Sin embargo, se puede ver la abstracción, y el concepto relacionado de

herencia, como algo más que simplemente un "muro de información" que impide al programador ver el contenido de objetos de datos no apropiados.

La herencia proporciona el mecanismo para pasar información entre objetos en clases relacionadas. Si $A \Rightarrow B$ significa que B es una clase relacionada con A, ¿cuál es la relación entre objetos de A y objetos de B? Existen cuatro relaciones de este tipo, como se muestra en la figura 8.9; sintetizan los diversos usos de la herencia en los lenguajes.

Especialización. Ésta es la forma más común de herencia, la cual permite que el objeto derivado B obtenga propiedades más precisas que las que están presentes en el objeto A [figura 8.9(a)]. Así, una *pila* es más precisa que los datos *compuestos*, y una *pila ent* es más precisa que una *pila*, aunque conserva las propiedades esenciales de una pila al heredar los métodos *push* (*agregar*) y *pop* (*remover*). Previamente se ha descrito *PilaNueva* como derivada de *PilaElem* por adición de la función *atisbar*. Se dice que la clase *PilaNueva* es una especialización de la clase *PilaElem*.

El concepto opuesto a la especialización es la *generalización* (por ejemplo, una *pila* es más general que una *pila ent* o una *pila real*). La generalización representa la superclase de un conjunto de subclases derivadas.

Descomposición. Es el principio que permite separar una abstracción en sus componentes [figura 8.9(b)]. Es el mecanismo típico de encapsulamiento en lenguajes como Ada sin la herencia de métodos. Por ejemplo, una *pila* (externamente) se compone de una variable llamada tope y un arreglo *almacenamiento*[100] internamente. los nombres internos no son conocidos fuera de la clase que define. El concepto contrario a descomposición es la *agregación*.

Ejemplarización. Es el proceso de crear ocurrencias de una clase. Es básicamente una operación de *copia* [figura 8.9(c)]. La declaración de ocurrencias de un objeto de clase en un programa en C++ es un ejemplo representativo de ejemplarización, y es el proceso de declarar el objeto x de tipo y en un programa:

```
stack A, B, C;
```

Esto causa que el entorno de ejecución haga tres copias del objeto de tipo *stack* (pila) y las nombre A, B y C.

El concepto opuesto a ejemplarización es la *clasificación*. En nuestro ejemplo, A, B y C se clasificarían como ocurrencias de la clase *stack*.

Individualización. Es el cuarto principio de abstracción [figura 8.9(d)], y es un poco más complejo de entender. En este caso, los objetos similares se agrupan unos con otros para propósitos comunes. El concepto inverso es el de *agrupamiento*.

Por ejemplo, entre las pilas de un traductor puede haber registros de activación y tablas de símbolos. Tanto C como Pascal implementan los registros de activación como pilas, pero se trata de pilas con características distintas. Todos estos son ejemplos de especialización. Todos tienen las mismas características esenciales ejemplificadas por las características de *push* y *pop* de una pila; sin embargo, el papel que un registro de activación en C o en Pascal desempeña en la ejecución del programa es diferente, no obstante que las estructuras subyacentes son similares.

8.3 POLIMORFISMO

El uso de parámetros para subprogramas es una de las características más antiguas de los lenguajes de programación; no obstante, en casi todos los lenguajes, los parámetros tienen una característica importante: tienen un valor l. Es decir, son un objeto de datos que requiere almacenamiento en el registro de activación del programa en tiempo de ejecución. Resulta muy familiar escribir *P*(*A*, 7) o *Q*(*B*,*false*, '*i*'). Sin embargo, los enunciados como *R*(*integer*, *real*) rara vez aparecen, donde los parámetros se refieren a tipos del lenguaje.

Polimorfismo significa la capacidad de un solo operador o nombre de subprograma para referirse a varias definiciones de función, de acuerdo con los tipos de datos de los argumentos y resultados. Ya se ha analizado una forma limitada de polimorfismo en el capítulo 4, cuando se estudió la homonimia de ciertos operadores. En las expresiones 1 + 2 y 1.5 + 2.7, se hace referencia a dos operadores + diferentes, un operador de enteros en la primera y un operador de números reales en la segunda. El uso de **generic** en Ada permite una homonimia limitada de las funciones, donde se compilan múltiples ocurrencias de una función, uno para cada uno de los tipos indicados que se destinan a los argumentos de función. La unificación en Prolog también se puede ver como una forma limitada de polimorfismo, cuando Prolog intenta unificar una consulta con el conjunto de reglas y hechos de la base de datos de Prolog.

El polimorfismo se aplica en general a funciones donde un tipo es uno de los argumentos. De los lenguajes que se estudian aquí, ML y Smalltalk son los que ofrecen la aplicabilidad más general del polimorfismo. La función de identidad, por ejemplo, puede aceptar argumentos de cualquier tipo. Si se define *ident* como:

```
fun ident(x) = x;
```

se obtiene esta respuesta de ML:

```
- fun ident(x) = x;
val ident = fn : 'a -> 'a
```

la cual significa que *ident* es una función que toma un argumento de tipo '*a* y devuelve un argumento de tipo '*a* para el parámetro de tipo '*a*. Por tanto, el paso de argumentos enteros, de cadena, y de lista de enteros a ident es válido en todos los casos:

```
- ident(3);
val it = 3 : int
- ident("abc");
val it = "abc" : string
- ident([1,2,3]);
val it = [1,2,3] : int list
```

En ML, ciertos operadores en una definición de función permiten polimorfismo, como la igualdad y los operadores que hacen listas a partir de objetos. Sin embargo, ciertos operadores restringen el dominio. Por ejemplo, los operadores aritméticos como + y – limitan los resultados a tipos aritméticos. Sin embargo, como recién se ha explicado, + ya tine un homónimo

como uno de los tipos de datos numéricos. Por esa razón, las definiciones en ML, como la que se dio anteriormente:

```
fun área(longitud,ancho) = longitud*ancho
```

son ambiguas, no obstante que *área*(1, 2) o *área*(1.3, 2.3) serían llamadas no ambiguas a la función. ML aplica una prueba estática sobre los tipos de argumentos de función, aunque una prueba dinámica de los tipos podría permitir polimorfismo para la función *área* ya citada.

La diferencia entre polimorfismo en ML y la función genérica de Ada es que Ada compilaría funciones *ident* individuales, una para manejar cada tipo distinto de argumento. En ML, basta con una sola ocurrencia de la función. En general, las funciones que emplean igualdad (o desigualdad) y creación de tuplas y listas en ML pueden ser polimórficas, puesto que estas operaciones no necesitan actuar sobre los datos reales mismos.

Los argumentos para funciones en ML pueden ser de cualquier tipo (que se determine estáticamente). Por tanto, se pueden escribir funciones polimórficas que toman argumentos de función y los aplican a objetos de datos. Considérense las dos funciones en ML siguientes: *longitud* calcula la longitud de una lista de enteros, e *imprímelo* imprime los elementos de la lista de enteros en orden:

```
fun longitud(nil) =          0
  | longitud(a::y) =         1+longitud(y);
fun imprímelo(nil) =         print("\n")
  | prx(a::y) int list) =    (print(a);imprímelo(y));
```

La función *proceso* se puede invocar ahora con argumentos de cualquier tipo, en tanto sean compatibles:

```
fun proceso(f,l) = f l;
```

donde *f l* es equivalente a *f*(*l*).

```
proceso(imprímelo,[1,2,3]);
123
proceso(longitud,[1,2,3]);
3
```

8.4 LECTURAS ADICIONALES SUGERIDAS

Las contribuciones que hizo David Parnas influyeron en favor del desarrollo de tipos de datos encapsulados a principios de la década de 1970 [PARNAS 1972]. La Conferencia de la ACM de 1977 *Language Design for Reliable Software* fue probablemente la reunión más importante donde se analizaron propuestas para mecanismos de encapsulamiento en lenguajes como Alphard, CLU, Gypsy, Mesa, Euclid y otros [WORKMAN 1977].

Capítulo 9

Avances en el diseño de lenguajes

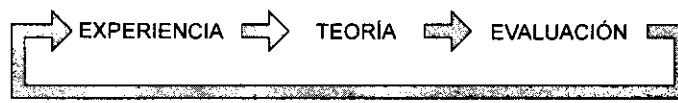
En los capítulos anteriores de este libro hemos presentado las características de los lenguajes de programación que son predominantes en la actualidad. Todos los lenguajes que implementan estas características siguen básicamente el modelo de la computadora de von Neumann que se describió en el capítulo 2. Para recordar ese diseño básico, diremos que una computadora se compone de una memoria principal relativamente grande, una memoria de control mucho más rápida pero más pequeña, y una unidad central de procesamiento que ejecuta instrucciones usando datos ya sea de la memoria de control o de la memoria principal.

En este capítulo estudiamos cómo están evolucionando los lenguajes. Primero analizamos paradigmas alternativos de programación que permiten usar los programas en una amplia variedad de computadoras. Luego examinamos cómo se están aplicando los métodos formales para ayudarnos a entender el proceso de programación, y por último veremos cómo está evolucionando la arquitectura de computadoras (tanto del *hardware* como del *software* que se ejecuta en el hardware) para encauzar nuestro endendimiento cambiante de nuestras necesidades de software.

En este capítulo estudiaremos los temas siguientes:

Variaciones del diseño de subprogramas. La activación de procedimiento **call-return** (llamada/regreso) (del capítulo 7) con vinculación estática y dinámica a objetos de datos tenía como base reglas de alcance, las cuales permiten implementar muchos lenguajes de programación actuales. Sin embargo, para tomar providencias en cuanto a características adicionales en nuestra estructura de subprogramas, como corrutinas, excepciones y tareas, examinaremos variaciones de este mecanismo básico.

Métodos formales del proceso de programación. La civilización moderna se basa en el trabajo conjunto de la ciencia y la ingeniería para ayudar a explicar la naturaleza y hacer posible el desarrollo de nuevas técnicas que servirán como instrumentos para adelantos futuros. El modelo general es:



Experimentamos la cruda realidad (por ejemplo, los programas tienen errores, la programación es difícil), desarrollamos teorías de diseño de lenguajes (por ejemplo, programación estructurada, gramáticas BNF, abstracciones de datos), evaluamos esas teorías (por ejemplo,

construimos lenguajes usando esos conceptos), las que conducen a nuevas mejoras (por ejemplo, desarrollo de clases orientadas a objetos como un avance respecto a las abstracciones de datos, desarrollo de las técnicas de análisis sintáctico SLR(1) y LALR(1) en lugar del análisis sintáctico LR(1)), los cuales mejoran aún más el proceso global. Hasta ahora hemos analizado en general algunas de las técnicas que han encontrado acomodo en el diseño de lenguajes. En este capítulo describiremos en forma breve algunos de los otros modelos teóricos que afectan el diseño de lenguajes de programación.

Cambios en las arquitecturas de hardware y software. Aunque las máquinas se vuelven más rápidas año con año, aproximadamente por un factor de dos cada tres años, el problema básico permanece. La memoria de control y la unidad central de procesamiento son siempre alrededor de un orden de magnitud más rápidas que la memoria principal más grande. Por consiguiente, es frecuente que una computadora consuma gran parte del tiempo simplemente esperando que el hardware introduzca datos desde la memoria principal mayor en la memoria de control, que es más rápida.

Se han intentado dos enfoques para resolver este problema. En un caso, se ha proyectado software que permite un uso más eficiente del hardware. Los enunciados como el **and**, que se analizará en la sección 9.2.1, permiten a los programadores codificar programas concurrentes. Esto hace posible que el hardware trabaje con más eficiencia ejecutando otro programa cuando un programa está bloqueado esperando datos. Hasta ahora, sin embargo, los programadores no han hecho un buen trabajo en cuanto a ser capaces de reescribir programas para que utilicen de manera automática estas nuevas mejoras a los lenguajes. La comprensión del comportamiento de ejecución de un programa dado es demasiado compleja como para aprovechar verdaderamente esas adiciones a los lenguajes.

La alternativa es desarrollar hardware más eficaz. La construcción de unidades centrales de procesamiento con memorias de control más grandes limita el número de veces que se debe tener acceso a la memoria principal para recuperar datos. El uso de una memoria caché entre la memoria principal y la memoria de control proporciona un aceleración efectiva de la memoria de control. Sin embargo, no importa cuán rápidas se hagan las computadoras, constantemente se están proyectando aplicaciones que requieren aún más recursos de la computadora.

Además, también está cambiando la naturaleza del cómputo. Las computadoras ya no se alojan en sus templos independientes. En los escritorios existen millones de poderosas estaciones de trabajo. La composición de este libro se hizo en una computadora personal conectada a la red internacional de computadoras Internet. El concepto de cómputo cliente/servidor, donde las tareas de computación se distribuyen entre un conjunto de computadoras está cambiando la naturaleza del software. Aunque la sección 9.4 hace énfasis en que se desea verificar la corrección y terminación de un programa, para muchas aplicaciones (por ejemplo, la red telefónica, un sistema de reservaciones de líneas aéreas u hoteles), el programa *nunca* debe terminar. La terminación costaría al usuario del programa una cantidad significativa en ingresos perdidos.

Con el fin de tratar de estas cuestiones, describiremos en forma breve algunas de las tendencias que están surgiendo actualmente en computación, así como sus efectos que, pensamos, influirán sobre el diseño de lenguajes. Separaremos estas tendencias en desarrollos de hardware y de software.

9.1 VARIACION DEL CONTROL DE LOS SUBPROGRAMAS

En esta sección se examinarán los cambios a la estructura usual de registros de activación para vinculación de subprogramas. Primero, recuérdese que la vinculación de subrutinas descrita en la sección 7.1 se basaba en cinco supuestos:

1. Los subprogramas no pueden ser recursivos.
2. Se requieren enunciados **call** explícitos.
3. Los subprogramas se deben ejecutar cabalmente en cada llamada.
4. Transferencia inmediata del control en el punto de llamada.
5. Una sola secuencia de ejecución.

Ya se ha examinado el primer supuesto, la recursión. Modifiquemos los otros supuestos e investiguemos la forma de mecanismo de subrutinas que resulta.

9.1.1 Excepciones y manejadores de excepciones

Durante la ejecución de un programa, suelen ocurrir sucesos o condiciones que se podrían considerar como "excepcionales". En vez de continuar con la ejecución normal del programa, se necesita llamar un subprograma para llevar a cabo cierto procesamiento especial. Esto incluye:

Condiciones de error que obligan a procesar un error, como exceder la capacidad de una operación aritmética o hacer referencia a un elemento de arreglo con un subíndice fuera de límites.

Condiciones impredecibles que surgen durante la ejecución normal del programa, como la producción de encabezamientos especiales de salida al final de una página de impresora o un indicador de final de archivo en un archivo de entrada.

Rastreo y monitoreo durante la prueba de programas, como la salida de rastreo de impresión durante la prueba de programas cuando el valor de la variable cambia.

Aunque por lo común es posible insertar una prueba explícita en el programa para probar en busca de estas condiciones excepcionales, estos enunciados adicionales pueden ocultar la estructura básica del programa. Es más sencillo relajar el requisito de que los subprogramas se deben llamar por medio de llamadas explícitas y suministrar una forma de invocar un subprograma cuando ocurre una condición o suceso particular. A una condición o suceso de esta clase se le suele llamar *excepción* o *señal*, y al subprograma que lleva a cabo el procesamiento especial se le conoce como *manejador de excepciones*. A la acción de advertir la excepción, interrumpir la ejecución del programa y transferir el control al manejador de excepciones se le describe como *plantear la excepción*. En Ada, existe una clase de excepciones llamadas *checks* (verificaciones) o condiciones que requieren que se ejecute código en tiempo de ejecución. Por ejemplo, *Index-Check* es una excepción de *Constraint_Error* (Error_De_Restricción) que se plantea cuando un límite de arreglo no es válido.

Manejadores de excepciones

Puesto que un manejador de excepciones se invoca sin una llamada explícita, ordinariamente no requiere un nombre o parámetros. La definición de un manejador de excepciones contiene típicamente sólo:

1. Un conjunto de declaraciones de variables locales (en su caso), y
2. Una serie de enunciados ejecutables.

Para suministrar la conexión entre las excepciones y sus manejadores, a cada clase de excepciones se le da un nombre. Ciertas excepciones pueden estar predefinidas en el lenguaje, por ejemplo *Constraint_Error* (Error_de_restricción), *Program_Error* (Error_De_Programa) o *Numeric_Error* (Error_Numérico) en Ada. Otras pueden ser definidas por el programador, por ejemplo, el programa puede incluir una declaración "Subdesbordamiento: **exception**" o "Desbordamiento: **exception**". Cada manejador de excepciones se aparea entonces con el nombre (o nombres) de la excepción o excepciones que va a manejar. Por lo común, todos los manejadores de excepciones se agrupan al principio o al final del programa o subprograma mayor donde la excepción podría ocurrir. Esta estructura en Ada es típica:

```
procedure Sub is
   Valor_De_Dato_Malo: exception;
      - otras declaraciones para Sub
begin
      - enunciados para procesamiento normal de Sub
exception
   when Valor_De_Dato_Malo =>
      - manejador para valores de datos malos
when Constraint_Error =>
      - manejador para excepción predefinida Constraint_Error
   when others =>
      - manejador para todas las demás excepciones
end;
```

Planteamiento de una excepción. Una excepción puede ser planteada mediante una operación primitiva definida por el lenguaje; por ejemplo, una operación de adición o multiplicación podría plantear la excepción Constraint_Error. Alternativamente, el programador podría plantear de manera explícita una restricción usando un enunciado suministrado para ese fin, como éste en Ada:

```
raise Valor_De_Dato_Malo;
```

el cual se podría ejecutar en un subprograma después de determinar que una variable o archivo de entrada contiene un valor impropio.

En un subprograma, si se usa un enunciado **raise** explícito y el subprograma mismo contiene un manejador para la excepción planteada, como por ejemplo cuando el enunciado:

```
if X = 0 then raise Valor_De_Dato_Malo end if;
```

aparece dentro del cuerpo del procedimiento *Sub* antes citado, entonces el enunciado **raise** tiene el efecto de transferir el control al manejador asociado, el cual abandona después el procedimiento. Uno de los usos principales de los enunciados **goto** en lenguajes sin

características explícitas de manejo de excepciones es proporcionar transferencias a código para manejo de excepciones, como se expone en la sección 6.4.2.

Se pueden pasar datos al manejador de excepciones, como lo demuestra el ejemplo siguiente en ML. La acción normal en ML es terminar el procedimiento, pero si se suministra un manejador, éste se ejecuta y el control regresa al punto del error, como en:

```
exception DenominadorMalo of int;
fun InnerDivide(a:int,b:int):real=
    if b=0 then raise DenominadorMalo(b)
        else real(a)/real(b);
fun Divide(a,b)= InnerDivide(a,b) handle
    DenominadorMalo(b) => (print(b); "es denominador malo, se usó 0"; 0.0);
```

En este caso, el programa invoca la función *Divide* (Dividir), la cual llama la función *InnerDivide* (DividirInterior). *InnerDivide* invoca la ejecución de la operación misma de dividir para ser manejada por la excepción DenominadorMalo. En este caso, el manejador imprime un mensaje y regresa el valor real 0.0 para la función *Divide* y la ejecución continúa. (Adviértase que no es necesario especificar tipos para los argumentos y el valor regresado de la función *Divide*. ML intenta inferir el tipo correcto, y en este caso puede hacerlo a partir de los tipos explícitos dados para *InnerDivide*.)

En C++ las excepciones se procesan (aunque no todo traductor de C++ tiene excepciones implementadas) a través de la cláusula **try**. C++ plantea una excepción *lanzando* la excepción, y la excepción se maneja *atrapando* la excepción. La sintaxis es similar a la de ML, aunque, a diferencia del ML, la ejecución se detiene después de manejar una excepción:

```
try {
    enunciado1;
    enunciado2;
    ...
    if CondiciónMala { throw NombreDeExcepción};
        }
catch NombreDeExcepción { // Hacer algo por la excepción
        } / / Fin de Excepción
    }
```

Propagación de una excepción. Cuando se construye un programa, con frecuencia el lugar donde ocurre una excepción no es el mejor lugar para manejarla. Por ejemplo, un subprograma puede tener la función de leer valores de datos de un archivo y pasarlos a un nido de subprogramas para su procesamiento. Supóngase que se pueden encontrar varios tipos distintos de valores de datos malos en un archivo y que cada subprograma prueba en busca de una clase diferente de estos errores, pero la respuesta es en todos los casos la misma: imprimir un mensaje de error y adelantar el archivo más allá de los datos malos. En este caso, el manejador

podría correctamente ser parte del subprograma que lee el archivo, y cada subprograma podría plantear adecuadamente la excepción *Valor_De_Dato_Malo*. Cuando una excepción se maneja en un subprograma distinto del subprograma en el cual se planteó, se dice que la excepción se ha *propagado* desde el punto en el cual se planteó hasta el punto donde se maneja.

La regla para determinar cuál manejador se encarga de una excepción particular se define por lo común en términos de la *cadena dinámica* de activaciones de subprograma que conduce al subprograma que plantea la excepción. Cuando una excepción *P* se plantea en el sub-programa *C*, entonces *P* es manipulada por un manejador definido en *C*, si es que existe uno. Si no lo hay, entonces *C* termina. Si el subprograma *B* llamó a *C*, entonces la excepción se propaga a *B* y se plantea de nuevo en el punto en *B* donde *B* llamó a *C*. Si *B* no suministra un manejador para *P*, entonces *B* se termina, la excepción se propaga al que llamó a *B*, y así sucesivamente. Si ningún subprograma ni el programa principal suministran un manejador, entonces el programa completo se termina y se invoca un manejador estándar definido por el lenguaje. Por ejemplo, en el ejemplo anterior en ML, la función *InnerDivide* planteó la excepción *Denominador_Malo*, la cual se propagó al manejador de la función *Divide*, misma que devolvió entonces un valor definido por el programador como resultado de manejar esa excepción.

Un efecto importante de esta regla para propagar excepciones es que permite que un subprograma permanezca como una *operación abstracta* definida por el programador, incluso al procesar excepciones. Una operación primitiva o un subprograma puede interrumpir súbitamente su procesamiento normal y plantear una excepción. Para el que llama, el efecto de que un subprograma plantee una excepción es precisamente el mismo que el efecto del planteamiento de una excepción por parte de una operación primitiva, si el subprograma no maneja él mismo la excepción. Si la excepción se maneja dentro del subprograma, entonces el mismo regresa en la forma normal y el que llama nunca se da cuenta de que se ha planteado una excepción.

Después de que se ha manejado una excepción. Después de que un manejador completa el procesamiento de una excepción, existe una duda constante en cuanto a dónde se va a transferir el control porque no hubo una llamada explícita del manejador. ¿Debe volver el control al punto donde se planteó la excepción (el cual puede estar a varios niveles de subprogramas de distancia)? ¿Debe volver el control al enunciado del subprograma que contiene el manejador donde la excepción se planteó después de propagarse? ¿Debe terminarse el subprograma que contiene el manejador mismo, pero terminarse de manera normal, para que parezca al que lo llamó como si nada hubiera ocurrido? Esta última solución es la que se adopta en Ada; ML proporciona varias opciones y otros lenguajes han elegido alternativas distintas.

Implementación

Las excepciones provienen de dos fuentes: (1) condiciones que detecta la computadora virtual, y (2) condiciones que genera la semántica del lenguaje de programación. En el primer caso, las excepciones de sistema operativo pueden ser planteadas directamente por interrupciones o trampas de hardware, como el desbordamiento aritmético, o se pueden plantear en software de apoyo, como una condición de final de archivo. En C, el programador tiene

acceso directo a estas señales que procesa el sistema operativo. El programador puede *habilitar* una interrupción, por ejemplo, la función *sigaction* en Unix, la cual especifica un procedimiento que se invoca cuando se plantea la señal dada.

El lenguaje de programación puede contar con providencias para excepciones adicionales haciendo que el traductor del lenguaje inserte instrucciones adicionales en el código ejecutable. Por ejemplo, para detectar la excepción Index-Check causada por un subíndice de arreglo demasiado grande o demasiado pequeño, el traductor inserta una serie explícita de instrucciones en cada referencia a un arreglo, como $A[I, J]$, para determinar si los valores de I y J están dentro de los límites declarados. Por tanto, a menos que el hardware o el sistema operativo suministre la verificación de excepciones, la verificación en busca de una excepción requiere cierta simulación por software. El costo de esta verificación de software, tanto en almacenamiento de código como en tiempo de ejecución, suele ser grande. Por ejemplo, puede tomar más tiempo llevar a cabo la verificación de límites de subíndices en $A[I, J]$ que el que se requiere para tener acceso al elemento del arreglo. A causa de este costo adicional, casi todos los lenguajes proporcionan un medio para desactivar la verificación en busca de excepciones en partes del programa donde el programador determina que esto no es peligroso [por ejemplo, **pragma** *Supress*(*Index.Check*) en Ada].

Una vez que se plantea la excepción, la transferencia de control a un manejador en el mismo programa se implementa comúnmente por medio de un salto directo al inicio del código del manejador. La propagación de excepciones de regreso por la cadena dinámica de llamadas de subprograma puede hacer uso de la cadena dinámica formada por los puntos de devolución de los registros de activación de subprograma en la pila central, como ya se ha expuesto. Al avanzar hacia arriba por la cadena dinámica, se debe terminar cada activación de subprograma usando una forma especial de instrucción **return** tanto para devolver el control al que llama como para plantear de nuevo la excepción en el mismo. La serie de devoluciones continúa hacia arriba de la cadena dinámica hasta que se alcanza un subprograma que tiene un manejador para la excepción planteada.

Una vez que se encuentra el manejador apropiado, se le invoca como en una llamada de subprograma ordinaria. Sin embargo, cuando el manejador termina, también puede terminar el subprograma que lo contiene, lo cual conduce a dos devoluciones normales de subprograma, una inmediatamente después de la otra. Cuando la cadena dinámica se ha desenrollado en esta forma hasta una devolución normal a un subprograma, ese subprograma continúa su ejecución en la forma usual.

Asertos

El concepto de aserto está relacionado con las excepciones. Un aserto es simplemente un enunciado que implica una relación entre objetos de datos en un programa, como el siguiente en C++:

```
#include < assert.h >
...
assert (x>y+1);
```

La macro predefinida en C++ genera simplemente:

```
if (x>y + 1) { /* Imprimir mensaje de error */ }
```

Es una forma eficaz de probar en cuanto a errores sin que codificaciones complejas abarroten el diseño del programa fuente. Después de desarrollar el programa, el **assert** puede permanecer

como documentación dentro del programa (véase la sección 9.4.2) y la macro se puede alterar para que no genere enunciados de programa y permanezca simplemente como un comentario.

9.1.2 Corrutinas

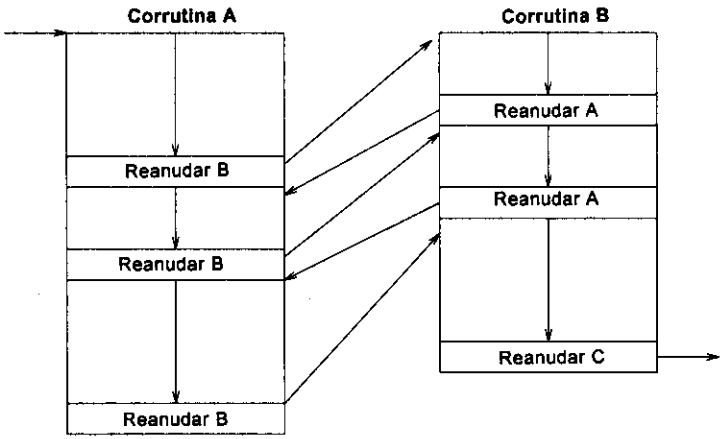


Figura 9.1. Transferencia del control entre corrutinas.

Supóngase que se quita la restricción 3 de la sección 9.1 para permitir que los subprogramas regresen a su programa de llamada antes de que se complete su ejecución. Estos subprogramas se llaman *corrutinas*. Cuando una corrutina recibe el control de otro subprograma, se ejecuta parcialmente y luego se suspende cuando regresa el control. En un punto posterior el programa que llama puede *reanudar* la ejecución de la corrutina a partir del punto en el cual se suspendió previamente la ejecución.

Adviértase la simetría que se ha introducido ahora en la estructura del programa de llamada y del llamado. Si *A* llama el subprograma *B* como corrutina, *B* se ejecuta un tiempo y regresa el control a *A*, justo como lo haría cualquier subprograma ordinario. Cuando *A* pasa de nuevo el control a *B* a través de un **resume** (reanudar) *B*, *B* se ejecuta otra vez por un tiempo y regresa el control a *A*, como un subprograma ordinario. Para *A*, por tanto, *B* parece un subprograma ordinario. Sin embargo, la situación es muy similar cuando se mira desde el subprograma *B*. *B*, a la mitad de la ejecución, *reanuda* la ejecución de *A*. *A* se ejecuta por un tiempo y regresa el control a *B*. *B* continúa su ejecución un tiempo y regresa el control a *A*. *A* se ejecuta por algún tiempo más y regresa el control a *B*. Desde el subprograma *B*, *A* se parece mucho a un subprograma ordinario. El nombre *corrutina* proviene de esta simetría. En vez de que se tenga una relación de progenitor a hijo o de llamador a llamado entre los dos subprogramas, ambos parecen más bien iguales, dos subprogramas que intercambian el control en un sentido y otro conforme cada uno se ejecuta y donde ninguno de los dos controla claramente al otro (figura 9.1).

Las corrutinas no son actualmente una estructura común de control en los lenguajes de programación, fuera de los lenguajes de simulación discreta (véase la sección 9.1.3). Sin

embargo, suministran una estructura de control en muchos algoritmos, la cual es más natural que la jerarquía ordinaria de subprogramas. Más aún, la estructura sencilla de corrutina se puede simular fácilmente en muchos lenguajes usando el enunciado **goto** y una variable de *punto de reanudación* que especifica la etiqueta del enunciado en el cual se debe reanudar la ejecución.

Implementación. La instrucción **resume**, que transfiere el control entre corrutinas, especifica la reanudación de cierta activación particular de la corrutina. Si existen múltiples activaciones recursivas de una corrutina *B*, entonces el enunciado **resume** *B* no tiene un significado claro. Por esta razón, es más sencillo pensar en las corrutinas en un contexto donde, cuando mucho, existe sólo una activación de una corrutina dada a la vez. Esta restricción permite utilizar una implementación para corrutinas que es similar a la que se usó para la estructura sencilla de llamada-regreso en la sección 6.4.2. A un solo registro de activación se le asigna almacenamiento en forma estática, al principio de la ejecución, como una extensión del segmento de código para la corrutina. En el registro de activación se reserva una sola localidad, que ahora se llama *punto de reanudación*, para guardar el antiguo valor de *pi* del CIP cuando una instrucción **resume** transfiere el control a otra corrutina. Sin embargo, a diferencia del punto de regreso en un subprograma sencillo, esta localidad del punto de reanudación en una corrutina *B* se usa para guardar el valor de *pi para B mismo*. La ejecución de una instrucción **resume** *B* en la corrutina *A* comprende por tanto dos pasos:

1. El valor actual del CIP se guarda en la localidad del punto de reanudación del registro de activación para *A*.
2. El valor de *pi* en la localidad del punto de reanudación de *B* se obtiene del registro de activación de *B* y se asigna al CIP para efectuar la transferencia del control a la instrucción apropiada en *B*.

Puesto que no hay una instrucción explícita de regreso, *B* no necesita saber que *A* le dio el control.

Las estructuras de control en las cuales los subprogramas se pueden invocar como corrutinas o como subprogramas ordinarios, y donde las corrutinas pueden ser recursivas (es decir, pueden tener múltiples activaciones simultáneas), requieren implementaciones más complejas.

9.1.3 Subprogramas planificados

El concepto de planificación de subprogramas es consecuencia del relajamiento del supuesto de que la ejecución de un subprograma siempre se debe iniciar de inmediato cuando se le llama. Se puede pensar que un enunciado ordinario de llamada de subprograma especifica que el subprograma llamado debe ponerse en ejecución de inmediato, sin completar la ejecución del programa de llamada. La finalización de la ejecución del programa de llamada se vuelve a planificar para que ocurra de inmediato al terminar el subprograma. La estructura de control para manejo de excepciones también se puede ver como un medio para cambiar la organización de los subprogramas. El manejador de excepciones se ejecuta siempre que se plantea una excepción particular.

Generalizando aún más, son posibles otras técnicas de reorganización de subprogramas:

1. Planificar los subprogramas para ejecutarse antes o después de otros subprogramas, como, por ejemplo: **call** *B* **after** *A*, que establecería la ejecución del subprograma *B* después de completar la ejecución del subprograma *A*.

2. Planificar los subprogramas para ser ejecutados cuando una expresión booleana arbitraria se hace cierta, como, por ejemplo:

   call B **when** $X = 5$ **and** $Z > 0$

   Esta organización proporciona una especie de característica generalizada para manejo de excepciones; *B* es llamado siempre que los valores de *Z* y *X* se cambian para satisfacer las condiciones dadas.

3. Planificar los subprogramas con base en una *escala de tiempo* simulada, como, por ejemplo:

   call B **at** time = 25 **or call** B **at** time = HoraActual + 10

   Esta planificación permite una intercalación general de llamadas de subprograma planificadas a partir de distintas fuentes.

4. Planificar los subprogramas de acuerdo con una designación de prioridad, como, por ejemplo:

   call B **with priority** 7

   que activaría a *B* cuando no ha sido planificado otro subprograma con prioridad más alta.

La planificación generalizada de subprogramas es una característica de los lenguajes de programación proyectados para simulación discreta de sistemas, como GPSS, SIMSCRIPT y SIMULA, aunque los conceptos tienen amplia aplicabilidad. Cada una de las técnicas de planificación anteriores aparece en al menos uno de los lenguajes de simulación mencionados. La técnica más importante para simulación de sistemas es la tercera de la lista: planificación con base en una escala de tiempo simulada. Se hará énfasis en esta técnica en nuestra exposición.

Cuando hablamos de planificación de subprogramas queremos decir planificación de *activaciones* de subprogramas, porque esta planificación es una actividad en tiempo de ejecución en la cual el mismo subprograma se puede organizar de modo que sea activado en muchos puntos distintos durante la ejecución. En la planificación generalizada de subprogramas, el programador ya no escribe un programa principal. En vez de ello, el programa principal es un *programa planificador* definido por el sistema y que mantiene típicamente una lista de activaciones de subprograma actualmente planificadas, las cuales se ordenan según la secuencia en la que se van a ejecutar. En el lenguaje se suministran enunciados a través de los cuales se pueden insertar activaciones de subprograma en esta lista durante la ejecución. El planificador opera llamando cada subprograma de la lista en el orden indicado. Cuando termina la ejecución de un subprograma, se inicia la ejecución del subprograma que sigue en la lista. Por lo común también se toman

disposiciones para llamadas de subprograma ordinarias, a veces simplemente permitiendo que un subprograma suspenda su propia ejecución y organice la ejecución inmediata de otro subprograma.

En los lenguajes de simulación, el enfoque más común en cuanto a planificación de subprogramas se basa en un tipo de corrutina generalizada. La ejecución de una sola activación de subprograma tiene lugar en una serie de fases *activas* y *pasivas*. Durante una fase activa el subprograma tiene el control y se está ejecutando; en una fase pasiva el subprograma ha transferido el control a otra parte y está aguardando una llamada de reanudación. Sin embargo, en vez de que cada corrutina transfiera el control directamente a otra corrutina cuando pasa de activa a pasiva, el control se regresa al planificador, el cual transfiere luego el control al subprograma que sigue en su lista de activaciones planificadas. Esta transferencia de control puede adoptar la forma de una llamada de reanudación si el subprograma ya está parcialmente ejecutado, o se puede iniciar una activación totalmente nueva del subprograma.

El concepto de planificación de corrutinas es particularmente directo cuando se usa una escala simulada de tiempo. Supóngase que cada fase activa de ejecución de un subprograma se puede planificar de modo que ocurra en cualquier punto de una escala de tiempo de enteros que comienza en el tiempo $T = 0$. T es una variable entera simple que siempre contiene el valor del tiempo presente en la escala simulada. La ejecución de una fase activa de un subprograma siempre tiene lugar de manera instantánea en esta escala de tiempo simulada; es decir, el valor de T no cambia durante la ejecución de una fase activa de un subprograma. Cuando un subprograma completa una fase activa y regresa el control al planificador, éste actualiza el valor de T al que corresponde al momento en que se debe activar el subprograma que sigue en la lista de subprogramas planificados y transfiere el control a ese subprograma. La rutina recién activada se ejecuta parcialmente y regresa el control al planificador, el cual actualiza T una vez más y activa la rutina siguiente de la lista.

9.1.4 Ejecución no secuencial

La última restricción de la sección 7.1 que hace falta examinar es la restricción a un solo orden de ejecución al ejecutar un programa. De manera más general, varios subprogramas se podrían estar ejecutando *simultáneamente*. Cuando hay un solo orden de ejecución, el programa se describe como un *programa secuencial*, porque la ejecución de sus subprogramas tiene lugar en un orden previamente definido. En el caso más general, el programa se describe como un programa *concurrente* o *en paralelo*. Todo subprograma que se puede ejecutar de manera simultánea con otros subprogramas se conoce como *tarea* (o a veces como *proceso*). En la sección siguiente se amplía este concepto.

9.2 PROGRAMACIÓN EN PARALELO

En las secuencias de control descritas hasta ahora, se ha supuesto que siempre existe una orla de control predeterminada que describe la ejecución de un programa. Esto se ajusta en gran medida a la arquitectura de von Neumann que se ha supuesto para la computadora real de hardware (sección 2.1.1). Sin embargo, los diseños en paralelo desempeñan un papel importante en programación.

Los sistemas de computadora capaces de ejecutar varios programas en forma simultánea son ahora bastante comunes. Un sistema de *multiprocesadores* tiene varias unidades centrales de procesamiento (CPU) que comparten una memoria común. Un sistema de computadoras *distribuidas* o *en paralelo* tiene varias computadoras (posiblemente cientos), cada una con su propia memoria y CPU, conectadas con vínculos de comunicación en una red en la cual cada una se puede comunicar con las otras. En esta clase de sistemas, se pueden ejecutar muchas tareas de manera simultánea.

Incluso en una sola computadora, suele ser útil proyectar un programa de manera que se componga de muchas tareas independientes que se ejecutan en forma concurrente en la computadora *virtual*, no obstante que en la computadora real sólo se puede estar ejecutando una en un momento dado. La ilusión de ejecución concurrente en un solo procesador se obtiene intercalando la ejecución de las tareas individuales, de manera que cada una ejecute una porción de su código, luego se intercambie para ser sustituida por otra tarea que ejecuta una porción de su código, y así sucesivamente. Los sistemas operativos que manejan *multiprogramación* y *tiempo compartido* proporcionan este tipo de ejecución concurrente para programas de usuario individuales. Lo que aquí nos ocupa, sin embargo, es la ejecución concurrente de tareas *dentro de un solo programa*.

El obstáculo principal es la carencia de construcciones para elaborar esta clase de sistemas. En la mayoría de los casos, los lenguajes normales como C se usan con llamadas de función adicionales al sistema operativo para habilitar tareas en paralelo. Sin embargo, el diseño del lenguaje de programación puede ayudar.

Principios de lenguajes de programación en paralelo

Las construcciones de programación en paralelo aumentan la complejidad del diseño del lenguaje, puesto que varios procesadores pueden estar teniendo acceso a los mismos datos de manera simultánea. Para estudiar el paralelismo en lenguajes de programación, es necesario tratar los cuatro conceptos siguientes:

1. *Definiciones de variables*. Las variables pueden ser *mutables* o *de definición*. Las variables mutables son las variables comunes que se declaran en casi todos los lenguajes secuenciales. Se pueden asignar valores a las variables y cambiarse durante la ejecución del programa. A una variable de definición se le puede asignar un valor una sola vez. La ventaja de esta clase de variable es que no existen problemas de sincronización. Una vez asignado un valor, cualquier tarea puede tener acceso a la variable y obtener el valor correcto.
2. *Composición en paralelo*. La ejecución avanza de un enunciado al siguiente. Además de los enunciados secuenciales y condicionales de los lenguajes de programación secuenciales, es necesario agregar el enunciado *en paralelo*, el cual causa que se comiencen a ejecutar *orlas de control* adicionales. El enunciado **and** de la sección 9.2.1 y la función *fork* (bifurcación) de sistema operativo que se llama desde C son ejemplos de esta clase de estructuras.
3. *Estructura del programa*. Los programas en paralelo se apegan en general a uno de los dos modelos de ejecución siguientes: (a) Pueden ser *transformativos* cuando la meta es transformar los datos de entrada en un valor de salida apropiado. El paralelismo se aplica para acelerar el proceso; por ejemplo, efectuar rápidamente la multiplicación de

una matriz multiplicando varias secciones de la misma en paralelo. (b) Pueden ser *reactivos* cuando el programa reacciona ante estímulos externos, llamados *sucesos*. Los sistemas en tiempo real y de comando y control son ejemplos de sistemas reactivos. Un sistema operativo y un sistema de procesamiento de transacciones, como un sistema de reservaciones, son ejemplos representativos de esta clase de sistemas reactivos. Se caracterizan por tener en general un comportamiento no determinista, puesto que nunca es explícito el momento exacto en que va a ocurrir un suceso.

4. *Comunicación.* Los programas en paralelo se deben comunicar unos con otros. Esta comunicación tiene lugar típicamente a través de *memoria compartida* con objetos de datos comunes a los que tienen acceso todos los programas en paralelo, o por medio de *mensajes*, donde cada programa en paralelo tiene su propia copia del objeto de datos y pasa valores de datos entre los otros programas en paralelo.

En las subsecciones siguientes, se analizan algunas de estas cuestiones con mayor detalle.

9.2.1 Ejecución concurrente

El mecanismo principal para instalar la ejecución en paralelo en un lenguaje de programación consiste en crear una construcción que dé lugar a ejecución en paralelo. El enunciado **and** lleva a cabo esta tarea.

enunciado$_1$ **and** *enunciado*$_2$ **and** ... **and** *enunciado*$_n$

y tiene la semántica que hace que cada uno de los diversos *enunciado*$_i$ se ejecuten en paralelo; el enunciado que sigue al enunciado **and** no se inicia hasta que terminan todos los enunciados en paralelo.

Aunque es sencillo desde el punto de vista conceptual, proporciona el pleno poder paralelo que se necesita para la ejecución en paralelo. Por ejemplo, si un sistema operativo consiste en una tarea de leer desde una terminal, una tarea de escribir en una pantalla y un proceso de ejecutar un programa de usuario, se podría especificar este sistema operativo como:

```
call LeerProceso and
call EscribirProceso and
call EjecutarProgramaDeUsuario;
```

Las secuencias para ejecución en paralelo son sólo parte del problema. El manejo correcto de los datos es otra preocupación. Considérese lo siguiente:

```
x:=1;
x:=2 and y := x+x;(*)
print(y);
```

Puesto que los dos enunciados rotulados con (*) se pueden ejecutar en paralelo, no es posible predecir cuál va a terminar primero. Por consiguiente, a y se le puede asignar el valor 2 (si la asignación a y se ejecuta primero), 4 (si la asignación a x se ejecuta primero), o incluso 3 (si

la asignación a *x* se hace entre los dos accesos de *x* en la asignación a *y*). Es necesario coordinar el acceso a datos en los programas concurrentes. Esto se analiza más a fondo al estudiar los *semáforos* en la sección 9.2.4.

Implementación. Existen dos formas básicas de implementar una construcción **and**. Adviértase que, si todas las tareas en paralelo se pueden ejecutar en paralelo, no existe ningún supuesto en cuanto a su orden de ejecución. Se podrían ejecutar simplemente en orden y, si la construcción **and** original es correcta, entonces el "reemplazo" de **and** por el ";" secuencial y un enunciado **while** daría lugar a una ejecución correcta. Por ejemplo, el compilador podría reescribir el ejemplo anterior como:

```
while MásPorHacer do
      MásPorHacer := false;
      call LeerProceso;
      call EscribirProceso;
      call EjecutarProgramaDeUsuario
end
```

Si, por ejemplo, el *LeerProceso* anterior intentara leer datos que todavía no estuvieran listos para ser procesados, podría simplemente fijar *MásPorHacer* como *cierto* y regresar. Esta iteración se repetirá en tanto cada subprograma no haya terminado todavía.

Una manera más directa de implementar esta construcción consiste en usar las primitivas de ejecución en paralelo del sistema operativo subyacente. Por ejemplo, en C, el compilador podría ejecutar una función *fork*, la cual tiene el efecto de crear dos procesos que se ejecutan en paralelo. Cada proceso continuaría ejecutándose hasta terminar. El código que genera el compilador de C sería algo así como:[1]

```
fork LeerProceso;
fork EscribirProceso;
fork EjecutarProgramaDeUsuario;
wait /* a que terminen los 3 */
```

Las facilidades para el manejo de tareas concurrentes son todavía poco comunes en lenguajes de programación de alto nivel. Ningún lenguaje de uso extendido da cabida a la construcción **and** que aquí se ha mencionado. De los lenguajes que se describen en la parte II, sólo Ada suministra tareas y ejecución concurrente, aunque la estrecha asociación de C con el sistema operativo permite que los programas en C invoquen la función *fork* de sistema operativo para crear tareas concurrentes.

9.2.2 Mandatos protegidos

Una segunda clase de construcciones tiene que ver con la ejecución no determinista, donde no se puede determinar cuál es el próximo enunciado por ejecutar. La construcción **and** anterior

[1] Este ejemplo es correcto sólo en parte para conservarlo sencillo. En realidad, C tendría que invocar primero la función *fork* para crear dos procesos y luego *exec* para llamar este proceso.

es en general determinista, puesto que sólo uno de los enunciados en paralelo se va a ejecutar efectivamente en seguida en una computadora dada; sin embargo, el concepto del **and** es la ejecución en paralelo, y su implementación suele ser un paso secuencial a través de cada uno de los enunciados.

El no determinismo auténtico fue propuesto a principios de los años setenta por Dijkstra con el concepto del *comando protegido* como medio para simplificar tanto el desarrollo de programas como su verificación [DJIKSTRA 1975]. Hasta aquí, todas las estructuras de control en este capítulo son deterministas. Ello significa que cada parte del enunciado tiene un orden específico de ejecución. Sin embargo, hay momentos en los cuales el no determinismo facilita el diseño de software, en forma muy parecida a la exposición con autómatas de estados finitos no deterministas como se hizo en la sección 3.3.2.

Una ejecución no determinista es aquella donde son posibles caminos de ejecución alternativos. Esta condición surge con frecuencia en la ejecución de programas concurrentes. Si existen *n* procesos listos para ejecutarse en un sistema, no siempre es claro cuál va a ser el siguiente que se ejecute. Si bien éste es un problema común en el diseño de sistemas operativos, no suele presentarse en el desarrollo de programas individuales, principalmente porque no se ha contado con las herramientas (por ejemplo, enunciados de lenguaje de programación) para pensar en formas no deterministas.

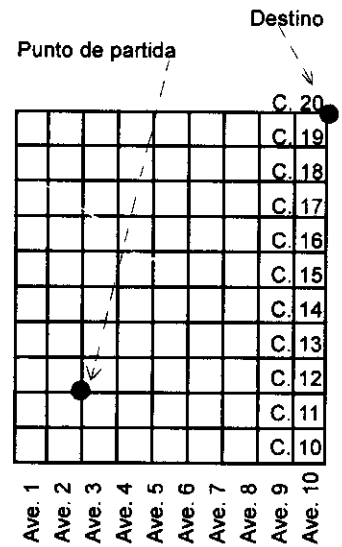


Figura 9.2. Recorrido no determinista en una ciudad.

Considérese como un ejemplo sencillo una ciudad compuesta de avenidas que corren de norte a sur y calles que corren de este a oeste (figura 9.2). Si una persona está en la Avenida Tres y la Calle Doce, ¿cuál es el camino para llegar a la Avenida Diez y la Calle Veinte?

Si se pide programar esto en un lenguaje convencional, surgirán típicamente dos diseños de programa distintos:

Diseño 1	**Diseño 2**
Ir al este de Ave. 3 a 10	*Ir al norte de Calle* 12 a 20
Ir al norte de Calle 12 a 20	*Ir al este de Ave.* 3 a 10

Sin embargo, son posibles muchos otros caminos. Si la ciudad tiene puntos de interés, entonces el transeúnte nunca verá los lugares de interés de la Avenida Ocho o de la Calle Dieciséis, por ejemplo. De hecho, para este programa se pueden especificar todos los caminos como:

while no en destino **do arbitrarily**
  if no en Ave. 10, ir al este 1 **o**
  if no en Calle 20, ir al norte 1
  end

Cualquier solución que siga esta ruta arbitraria (es decir, no determinista) es una solución válida del problema original. Pronto veremos que los comandos protegidos de Dijkstra son una solución general para este problema.

Guardas. El concepto básico es la *guarda*, la cual se escribe como : →. Si B es una *guarda* (es decir, condición) y S es un *comando* (es decir, un enunciado), entonces un comando protegido, $B \rightarrow S$, significa que el enunciado S está habilitado o listo para ejecutarse si la guarda B es *cierta*. Se usan guardas en los enunciados **guarded if** y **repetitive**:

Enunciado guarded if. Si B_i es un conjunto de condiciones y S_i es un conjunto de enunciados, el **guarded if** se escribe así:

$$\textbf{if } B_1 \rightarrow S_1 \,[\!]\, B_2 \rightarrow S_2 \,[\!]\, \ldots \,[\!]\, B_n \rightarrow S_n \textbf{ fi}$$

cuyo significado es que al menos una de las guardas *debe* ser cierta, y la ejecución es la de un comando correspondiente.

Adviértase que esto difiere de las construcciones anteriores que hemos estudiado: difiere de **if-then** normal, puesto que el caso usual no requiere que una guarda sea cierta, y no es lo mismo que el *cond* de LISP, donde el enunciado por ejecutar era el primero de la serie con una condición cierta. El **if** protegido está definido con auténtico no determinismo. Se elige arbitrariamente cualquier S_i para ejecutarlo en tanto B_i sea cierto.

Enunciado de repetición protegida. Se trata de una generalización del enunciado **while** secuencial y es similar al **guarded if**: si B_i es un conjunto de guardas y S_i es un conjunto de comandos, el **do** protegido se escribe así:

$$\textbf{do } B_1 \rightarrow S_1 \,[\!]\, B_2 \rightarrow S_2 \,[\!]\, \ldots \,[\!]\, B_n \rightarrow S_n \textbf{ od}$$

La ejecución tiene lugar como sigue: si algunas guardas B_i son ciertas, se ejecuta uno de los S_i correspondientes. Este proceso se repite en tanto alguna guarda sea cierta.

Si ninguna guarda es cierta al principio, el **do** se pasa por alto, lo cual es similar al comportamiento del enunciado **while** usual. Como en el caso del **if** protegido, se introduce el no determinismo en la ejecución si más de una guarda es cierta al mismo tiempo.

Los comandos protegidos suelen facilitar el desarrollo o la comprensión de muchos algoritmos. Considérese el problema anterior de caminar por la ciudad. Se puede especificar la solución general de manera no determinante como sigue:

```
Ave = 3; {En Avenida 3}
Calle = 12; {En Ave 12}
do  Ave < 10 → Ave = Ave + 1
  || Calle < 20 → Calle = Calle + 1 od
```

Todas las ejecuciones que comienzan en 3 × 12 y terminan en 10 × 20 son ejecuciones válidas de este programa.

Ningún lenguaje común ha implementado verdaderamente comandos protegidos según los definió Dijkstra; sin embargo, el concepto de ejecución no determinista surge en el diseño de software de sistemas operativos. El papel de esta clase de comandos protegidos en el uso de tareas y subprogramas se trata más cabalmente en la subsección siguiente.

9.2.3 Tareas

La idea básica que está atrás de las tareas es simple. Considérese un subprograma *A* que se está ejecutando en la forma normal. Si *A* llama el subprograma *B*, entonces ordinariamente la ejecución de *A* se suspende mientras *B* se ejecuta. Sin embargo, si *B* se inicia como una *tarea*, entonces la ejecución de *A* continúa mientras *B* se ejecuta. Ya sea *A* o *B*, o ambos, pueden iniciar ahora tareas adicionales, lo cual permite que coexista cualquier número de secuencias de ejecución en paralelo.

En general, cada tarea se considera como una *dependiente* de la tarea que la inició. Cuando una tarea está lista para terminar, debe esperar hasta que todas sus dependientes hayan concluido para que pueda finalizar. Así pues, la división en múltiples secuencias de ejecución se invierte conforme las tareas terminan, fundiéndose en cada vez menos secuencias hasta que finalmente sólo queda una secuencia. En circunstancias normales cada una de estas tareas de máximo nivel controla una parte importante del sistema (que actualmente suele ser un sistema de computadoras distribuidas) y, una vez iniciada, se espera que se ejecute "para siempre".

Gestión de tareas

La definición de una tarea en un programa difiere poco de la definición de un subprograma ordinario, excepto en cuanto a definir cómo se sincroniza la tarea y de qué manera se comunica con otros programas. La mayor parte del cuerpo de una definición de tarea contiene declaraciones y enunciados ordinarios que se ocupan del procesamiento que la tarea lleva a cabo mientras trabaja en forma independiente de otras tareas. En Ada, que es aquí nuestro ejemplo primordial, una definición de tarea toma la forma siguiente:

```
task Nombre is
      - Declaraciones especiales que permiten sincronización
      y comunicación con otras tareas
   end;
task body Nombre is
      - Declaraciones locales usuales según se encuentran en cualquier subprograma
   begin
      - Serie de enunciados
   end;
```

La iniciación de la ejecución de una tarea puede adoptar la forma de una llamada de subprograma ordinaria. Por ejemplo, en muchas implementaciones de PL/I, una tarea *B* se inicia ejecutando el enunciado:

call B (*parámetros*) **tarea**;

En Ada el método es algo distinto. La definición de tarea, como se dio anteriormente, se incluye entre las declaraciones de alguna estructura de programa más grande, como el programa principal. Cuando se entra a la estructura de programa más grande, todas las tareas declaradas dentro de la misma se inician en forma automática. Por tanto, no se necesita un enunciado **call** explícito; las tareas comienzan a ejecutarse simultáneamente tan pronto como se entra en la estructura de programa más grande.

Es frecuente que se requieran múltiples activaciones simultáneas de la *misma* tarea en aplicaciones. Por ejemplo, considérese un sistema de computadora que controla un conjunto de terminales de usuario. La tarea principal podría ser el programa que vigila la condición de todas las terminales. Cuando un usuario se identifica en una terminal, esta tarea, *Monitor*, inicia una nueva tarea, *Terminal*, para controlar las interacciones con el usuario de esa terminal particular. Cuando el usuario se retira, la tarea *Terminal* concluye. La tarea *Monitor*, desde luego, se ejecuta continuamente excepto en caso de una falla catastrófica del sistema. Cuando varios usuarios están en comunicación simultáneamente en terminales distintas, se requieren varias activaciones de la tarea *Terminal*, una para cada usuario.

Si una tarea se inicia usando una llamada de subprograma ordinaria, como en PL/I, entonces la ejecución repetida de un **call** basta para crear múltiples activaciones. En Ada, la definición de tarea antes descrita se puede usar para crear únicamente una sola activación de tarea, gracias a la iniciación implícita de tareas. Así, la tarea *Monitor* estaría definida probablemente como ya se ha indicado. Por lo que toca a la tarea *Terminal*, se requieren múltiples activaciones y éstas deben ser creadas e iniciadas por *Monitor* según sea necesario. En Ada, *Terminal* se define como un *tipo tarea*.

```
task type Terminal is
    - Resto de la definición en la misma forma que antes
end;
```

La definición de *Terminal* como un tipo de tarea permite tratar una activación de la tarea como un tipo de objeto de datos, en la misma forma en que una definición ordinaria de tipo se

usa para definir una clase de objetos de datos, como se describe en la sección 5.2. El establecimiento (es decir, creación e iniciación) de una nueva activación de tarea es entonces lo mismo que si un nuevo objeto de datos utiliza una definición de tipo como plantilla. Para crear varias activaciones y dar los nombres *A*, *B* y *C*, el programador en Ada escribe las declaraciones como declaraciones de variable ordinarias:

```
A: Terminal;
B, C: Terminal;
```

Estas declaraciones aparecen al principio de una estructura de programa más grande, y al entrar a este programa mayor se crean e inician las tres activaciones de *Terminal*. Alternativamente, se puede definir una variable apuntador cuyo valor es un apuntador a una *activación de tarea*, como en:

```
type AptTarea is access Terminal;          -Define tipo apuntador
TérminoNuevo: AptTarea := new Terminal;    -Declara variable apuntador
```

La variable *TérminoNuevo* apunta a una activación de una tarea de tipo *Terminal* que se crea en el momento en que se crea la misma *TérminoNuevo*.

Una vez que se inicia una tarea, los enunciados de su cuerpo se ejecutan en orden, igual que para un subprograma ordinario. Cuando una tarea termina, no regresa el control. Su secuencia individual de ejecución en paralelo simplemente concluye. Sin embargo, una tarea no puede terminar hasta que sus dependientes han terminado y, cuando termina, cualquier tarea de la cual sea dependiente deberá ser notificada de ello para que esa tarea también pueda terminar. Una tarea termina cuando completa la ejecución de los enunciados de su cuerpo; una tarea que nunca termina se escribe de modo que contenga una iteración infinita que avanza en ciclos continuamente (hasta que ocurre un error).

9.2.4 Sincronización de tareas

Durante la ejecución concurrente de varias tareas, cada una procede *asincrónicamente* respecto a las otras; es decir, cada tarea se ejecuta a su propia velocidad, en forma independiente de las demás. Así, cuando la tarea *A* ha ejecutado 10 enunciados, la tarea *B*, que se inició al mismo tiempo, puede haber ejecutado 6 enunciados o ninguno, o puede haberse ejecutado ya hasta su conclusión y estar terminada.

Para que dos tareas que se ejecutan asincrónicamente coordinen sus actividades, el lenguaje debe suministrar un medio de *sincronización*, de modo que una tarea puede avisar a otra cuando completa la ejecución de una sección particular de su código. Por ejemplo, una tarea puede estar controlando un dispositivo de entrada y la segunda tarea estar procesando cada lote de datos conforme es introducido desde el dispositivo. La primera tarea lee en un lote de datos, indica a la segunda que ha llegado un lote y luego inicia la preparación para introducir el lote siguiente de datos. La segunda tarea espera la señal de parte de la primera tarea, luego procesa los datos, después indica a la primera que ha completado el procesamiento y entonces espera otra vez la señal de que ha llegado otro lote. Las señales que se envían entre las tareas

permiten que éstas sincronicen sus actividades para que la segunda no comience a procesar datos antes de que la primera haya terminado de leerlos e introducirlos, y para que la primera no sobreescriba datos que la segunda todavía está procesando.

Las tareas que están sincronizando sus actividades en esta forma son un poco similares a corrutinas. Las señales sirven para decir a cada tarea cuándo debe esperar y cuándo seguir adelante, un poco como el uso de llamadas de reanudación entre corrutinas para indicar a una corrutina que debe proceder. Sin embargo, con las corrutinas sólo hay un orden de ejecución, mientras que aquí puede haber varios.

Interrupciones. La sincronización de tareas concurrentes a través del uso de interrupciones es un mecanismo común que se encuentra en hardware de computadoras. Si la tarea *A* desea indicar a la tarea *B* que ha ocurrido un suceso particular (por ejemplo, que se ha completado un segmento particular de código), entonces la tarea *A* ejecuta una instrucción que hace que la ejecución de la tarea *B* se interrumpa de inmediato. El control se transfiere a un subprograma o segmento de código cuyo único propósito es manejar la interrupción llevando a cabo las acciones especiales que se requieren. Cuando este manejador de interrupciones completa su ejecución, la tarea *B* continúa su ejecución a partir del punto donde ocurrió la interrupción. Este método de señalamiento es similar a los mecanismos de manejo de excepciones descritos en la sección 9.1.1 y suele usarse para ese fin. Por ejemplo, en una computadora de hardware, una tarea que maneja un dispositivo de entrada-salida se puede sincronizar con el procesador central a través del uso de interrupciones. Sin embargo, en los lenguajes de alto nivel las interrupciones tienen varias desventajas como mecanismo de sincronización: (1) el código para el manejo de interrupciones es independiente del cuerpo principal de la tarea, lo que conduce a una estructura de programa confusa; (2) una tarea que desea esperar una interrupción debe entrar comúnmente en una iteración *de espera ocupado*, una iteración que no hace otra cosa que describir ciclos de manera interminable hasta que ocurre la interrupción; (3) la tarea se debe escribir de tal manera que se pueda manejar correctamente una interrupción en cualquier momento, lo cual requiere por lo común que los datos compartidos entre el cuerpo de la tarea y la rutina de interrupción se protejan en formas especiales. A causa de estos (y varios otros) problemas con las interrupciones, los lenguajes de alto nivel suministran ordinariamente otros mecanismos de sincronización.

Semáforos. Un *semáforo* es un objeto de datos que se usa para sincronizar tareas. Se compone de dos partes: (1) un contador de enteros, cuyo valor es siempre positivo o cero, y que se usa para contar el número de señales enviadas pero todavía no recibidas, y (2) una cola de tareas que están esperando el envío de señales. En un *semáforo binario*, el contador sólo puede tener los valores cero y uno. En un *semáforo general*, el contador puede adoptar cualquier valor positivo entero.

Se definen dos operaciones primitivas para un objeto de datos semáforo *P*:

signal(P). Cuando es ejecutada por una tarea *A*, esta operación prueba el valor del contador en *P*; si es cero, entonces la primera tarea de la cola de tareas se quita de la fila y se reanuda su ejecución; si no es cero o si la cola está vacía, entonces el contador se incrementa en uno (lo que indica que se ha enviado una señal pero todavía no se ha recibido). En cualquier caso, la ejecución de la tarea *A* continúa después de que se ha completado la operación *signal* (señalar).

wait(P). Cuando es ejecutada por una tarea *B*, esta operación prueba el valor del contador en

```
task A:
begin
   — Introducir primer conjunto de datos
   loop
      signal(ComenzarB) — Invocar tarea B
      — Introducir conjunto de datos siguiente
      wait(ComenzarA) — Esperar a que
         Tarea B termine con los datos
   endloop;
end A;
```

```
task B;
begin
   loop
      wait(ComenzarB) — Esperar a que tarea
         A lea los datos
      — Procesar datos
      signal(ComenzarA) — Indicar
         a Tarea A que continúe
   endloop;
end B;
```

Figura 9.3. Sincronización de tareas usando *signal* (señalar) y *wait* (esperar).

P; si es diferente de cero, entonces el valor del contador se reduce en uno (lo que indica que *B* ha recibido una señal) y la tarea *B* continúa su ejecución; si es cero, entonces la tarea *B* se inserta al final de la cola de tareas para *P* y la ejecución de *B* se suspende (lo que indica que *B* está esperando el envío de una señal).

Tanto *signal* como *wait* tienen una semántica simple que requiere el principio de *atomicidad*; cada operación completa su ejecución antes de que cualquier otra operación concurrente pueda tener acceso a sus datos. La atomicidad impide que ocurran ciertas clases de sucesos no deterministas indeseables. Por ejemplo, si *átomo*(*S*) representa la ejecución atómica (es decir, que no se puede interrumpir) del enunciado *S*, el ejemplo no determinista de la sección 9.2.1 se puede reescribir como:

```
x:=1;
x:=2 and átomo(y := x+x);
print(y);
```

En este caso, el resultado puede ser 2 o 4, según cuál sea el enunciado que se ejecuta primero, pero se evita el resultado indeseable de 3 como respuesta.

Como ejemplo del uso de semáforos y de las operaciones *wait* y *signal*, considérense una vez más las dos tareas que cooperan para (1) introducir un lote de datos (tarea *A*) y (2) procesar un lote de datos (tarea *B*). Para sincronizar sus actividades, se podrían usar dos semáforos binarios. La tarea *A* utiliza el semáforo *IniciarB* para indicar que se ha completado la entrada de un lote de datos. El semáforo *IniciarA* es utilizado por la tarea *B* para señalar que terminó el procesamiento de un lote de datos. La figura 9.3 muestra la estructura de las tareas *A* y *B* con el uso de estos semáforos.

El uso de semáforos en lenguajes de programación de alto nivel tiene ciertas desventajas: (1) una tarea sólo puede esperar un semáforo a la vez, pero suele ser deseable permitir que una tarea espere cualquiera de varias señales; (2) si una tarea no *señala* en el punto apropiado (por ejemplo, a causa de un error de codificación), el sistema completo de tareas se puede *trabar*. Esto es, las tareas pueden estar cada una esperando en una cola de semáforos a que alguna otra tarea dé la señal, de manera que no quedan tareas en ejecución; y (3) los programas en los que intervienen varias tareas y semáforos se vuelven cada vez más difíciles de entender,

depurar y verificar. En resumen, el semáforo es una construcción de sincronización de nivel relativamente bajo que es adecuada principalmente en situaciones sencillas.

El semáforo tiene un inconveniente adicional en los ambientes actuales. Las semánticas de *signal* y *wait* implican que todas las tareas que tienen acceso al semáforo comparten memoria. Con el crecimiento de los sistemas de procesamiento múltiple y las redes de computadoras, esto no es necesariamente cierto. El uso de *mensajes* está relacionado con el semáforo, pero no está restringido a un espacio de memoria compartido.

Mensajes. Un mensaje es una transferencia de información de una tarea a otra. Proporciona un medio para que cada tarea sincronice sus acciones con otra tarea y sin embargo la tarea permanece libre para continuar ejecutándose cuando no necesita estar sincronizada. El concepto básico es similar a un tubo. Un comando *send* (enviar) coloca un mensaje en el tubo (o *cola de mensajes*), en tanto que una tarea que espera un mensaje emite un comando *receive* (recibir) y acepta un mensaje proveniente del otro extremo del tubo. La tarea que envía está libre para continuar ejecutándose, enviando más mensajes y llenando la cola de mensajes, mientras que la tarea receptora continuará ejecutándose en tanto existan mensajes pendientes en espera de ser procesados.

Por ejemplo, una aplicación típica es el problema de *productora - consumidora*. La tarea productora obtiene datos nuevos (como la lectura de entradas desde el teclado), en tanto que la tarea consumidora utiliza esos datos (como la compilación del programa recién tecleado). Si se concede que *enviar*(*a*, *mensaje*) signifique que la tarea está enviando el *mensaje* a la tarea *a*, y que *recibir*(*desde*, *mensaje*) signifique que la tarea está esperando que se coloque un mensaje en *mensaje* desde la tarea *desde*, se puede programar la tarea de la productora como sigue:

```
task Productora;
begin
   loop - mientras más por leer
        - Leer datos nuevos;
        enviar(Consumidora,datos)
   endloop;
end Productora
```

y la tarea de la consumidora como:

```
task Consumidora;
begin
   loop - mientras más por procesar
      recibir(Productora,datos);
      - Procesar datos nuevos
   endloop;
end Consumidora
```

Los mensajes son bastante versátiles. El problema 1 al final del capítulo investiga cómo se pueden usar mensajes para simular semáforos.

La implementación del paso de mensajes es más compleja de lo que podría parecer. Varias tareas pueden intentar simultáneamente enviar mensajes a una sola tarea receptora. Para que

no se pierdan algunos de estos mensajes, la implementación deberá incluir un mecanismo para guardar los mensajes en una cola (por lo común llamada *buffer*) hasta que el receptor los pueda procesar. Alternativamente, se puede hacer que la tarea misma que envía (en vez de enviar sólo el mensaje) espere en una cola hasta que la receptora esté en condiciones de recibir el mensaje. Este último método se usa en Ada, donde una tarea que envía debe encontrarse (véase más abajo) con la tarea receptora (y por tanto sincronizarse con ella) para que el mensaje se pueda transmitir.

Comandos protegidos. Como se describe en la sección 9.2.2, hemos definido el comando protegido como una forma de incorporar no determinismo en programación. La sincronización es una forma de no determinismo, puesto que no siempre queda claro cuál tarea es la siguiente por ejecutar. La guarda constituye un buen modelo de sincronización de tareas.

El comando **if** protegido en Ada se conoce como enunciado **select** (seleccionar) y tiene la forma general siguiente (Ada establece ciertas restricciones adicionales que no se mencionan aquí):

```
select
   when condición_1 => enunciado_1
   or when condición_2 => enunciado_2
   ...
   or when condición_n => enunciado_n
   else enunciado_n+1     Cláusula else adicional
   end select;
```

Como el **if** protegido, cada una de las condiciones se conoce como una *guarda*, y cada enunciado es un *comando*. Una de las guardas que se evalúa como cierta determina cuál enunciado se va a ejecutar en seguida. Aunque los comandos protegidos se pueden usar como parte de diversos mecanismos de sincronización de tareas, su uso en el mecanismo de encuentro de Ada ilustra bien el concepto.

Encuentros. Ada emplea un mecanismo de sincronización muy similar al mensaje, pero requiere una acción de sincronización con cada mensaje. Cuando dos tareas sincronizan sus acciones durante un periodo breve, esa sincronización se conoce como un *encuentro* en Ada. Supóngase que una tarea *A* se utiliza para introducir datos, como en el ejemplo anterior, y la segunda tarea *B* procesa los datos. Sin embargo, supóngase que *B copia* los datos en un área local para datos antes de procesarlos, para que *A* pueda introducir un nuevo lote sin esperar. Ahora se necesita un encuentro para permitir que *A* indique a *B* que está listo un nuevo lote de datos. La tarea *A* debe esperar entonces mientras *A* copia los nuevos datos en su área local, y luego ambas tareas pueden continuar en forma concurrente hasta que *A* haya introducido un nuevo lote de datos y *B* haya procesado el último lote, punto en el cual tiene lugar otro encuentro.

Un punto de encuentro en *B* se conoce como una *entrada*, la cual en este ejemplo se podría llamar *DatosListos*. Cuando la tarea *B* está en condiciones de comenzar a procesar un nuevo

lote de datos, deberá ejecutar un enunciado **accept** (aceptar):

```
accept DatosListos do
- enunciados para copiar datos nuevos desde A en el área
      local para datos de B
end;
```

Cuando la tarea *A* ha completado la introducción de un nuevo lote de datos, debe ejecutar la *llamada de entrada: DatosListos*. Cuando la tarea *B* alcanza el enunciado **accept**, espera hasta que la tarea *A* (o alguna otra tarea) ejecuta una llamada de entrada para la entrada *DatosListos* nombrada en el enunciado **accept**. En forma similar, cuando la tarea *A* llega a la llamada de entrada *DatosListos,* espera hasta que *B* alcanza el enunciado **accept**. Cuando ambas están en ese punto, tiene lugar el encuentro. *A* continúa esperando mientras *B* ejecuta todos los enunciados contenidos en el **do ... end** del enunciado **accept**. En ese momento el encuentro está completo y tanto *A* como *B* continúan sus ejecuciones por separado.

Para ver cómo se podrían usar los comandos protegidos para permitir que *B* aguarde cualquiera de los diversos encuentros, supóngase que *B* se amplía de modo que pueda procesar datos provenientes de cualquiera de tres dispositivos de entrada, cada uno controlado por tareas individuales, *A*1, *A*2 y *A*3. Cada una de las tres tareas de introducción se ejecuta en forma concurrente con B y con las demás. Cuando una tarea de introducción tiene un lote de datos listo para ser procesado, ejecuta la llamada de entrada correspondiente, *Listo*1 (en la tarea *A*1), *Listo*2 (en la tarea *A*2) y *Listo*3 (en la tarea *A*3). Cuando se emite una de estas llamadas de entrada, la tarea *B* puede estar ya esperando o estar todavía procesando un lote anterior de datos. Si *B* estuviera ya esperando, entonces, sin una estructura de comandos protegidos, *B* no podría aguardar cualquiera de *Listo*1, *Listo*2 o *Listo*3, sino que tendría que esperar la llegada de sólo una de esas llamadas. Para aguardar cualquiera de las tres, *B* ejecuta el comando protegido:

```
select accept Listo1 do
   - Copiar datos desde A1 en área local de B
   end;
or accept Listo2 do
   - Copiar datos desde A2 en área local de B
   end;
or accept Listo3 do
   - Copiar datos desde A3 en área local de B
   end;
end select;
```

Cuando *B* alcanza este enunciado, espera hasta que *A*1, *A*2 o *A*3 indique la llamada de entrada apropiada. La llamada de entrada es aceptada (si se indica más de una al mismo tiempo, sólo una se acepta) y el encuentro tiene lugar como antes. Adviértase que se han omitido aquí las guardas explícitas **when** *condición* => y la cláusula **else**, puesto que los tres enunciados **accept** deben estar disponibles para ejecución cuando se alcanza el enunciado **select**. En ciertos casos, sin embargo, se podría incluir una guarda explícita. Por ejemplo, cada dispositivo de entrada podría tener una "situación" asociada que indique si estaba operando correctamente. El encuentro se podría condicionar a la situación de cada dispositivo:

```
select
    when SituaciónDispositivo1 = ON => accept Listo1 do ... end;
or when SituaciónDispositivo2 = ON => accept Listo2 do ... end;
or when SituaciónDispositivo3 = conectado => accept Listo3
        do ... end;
else ...            - Ningún dispositivo está listo; hacer otra cosa
end select;
```

Tareas y procesamiento en tiempo real

De un programa que debe interactuar con dispositivos de entrada-salida u otras tareas dentro de cierto periodo fijo se dice que está operando en *tiempo real*. Los programas que vigilan el rendimiento de un automóvil, controlan un horno de microondas, e incluso gobiernan un reloj digital pueden tener que reaccionar en tiempos inferiores a 100 milisegundos para un rendimiento correcto. Así, por ejemplo, es posible que una tarea *A* que desea tener un encuentro con otra tarea *B* no pueda demorarse más de un tiempo fijo antes de seguir adelante, incluso sin iniciar el encuentro. En los sistemas de computadoras en tiempo real, la falla de parte del hardware de un dispositivo externo de entrada-salida suele conducir a que una tarea se termine abruptamente. Si otras tareas están esperando a esta tarea fallida, el sistema completo de tareas se puede trabar y causar la caída del sistema.

Las demandas especiales del procesamiento en tiempo real requieren que el lenguaje incluya alguna noción explícita de *tiempo*. En Ada hay un paquete definido por el lenguaje, llamado *Calendar* (Calendario) que incluye un tipo *Time* (Tiempo) y una función *clock* (reloj). Una tarea que espera un encuentro debe "mirar el reloj", como en:

```
select DatosListos;
or delay 0.5;       - Esperar cuando mucho 0.5 segundos
end select;
```

Éste es un comando protegido sin guardas explícitas, de modo que una u otra alternativa está disponible para ejecución, con lo cual se evita una caída del sistema si transcurren 0.5 segundos y *DatosListos* no está habilitado todavía.

Tareas y datos compartidos

Cada tratamiento distinto de las estructuras de control de subprogramas, por ejemplo, corrutinas, manejadores de excepciones, tareas y subprogramas planificados, según se expuso en la sección 9.1, conduce a estructuras algo diferentes para datos compartidos. En casi todos los casos, estas estructuras son variaciones sencillas sobre los conceptos presentados en las secciones anteriores. Existen dos cuestiones: (1) gestión de almacenamiento y (2) exclusión mutua.

Gestión de almacenamiento en tareas. Las tareas se definen como varias secuencias de ejecución que se ejecutan en forma simultánea dentro de un programa dado. Típicamente, esto se representa como varios conjuntos de procedimientos que se ejecutan de manera independiente y se comunican entre sí empleando algunos de los mecanismos de exclusión

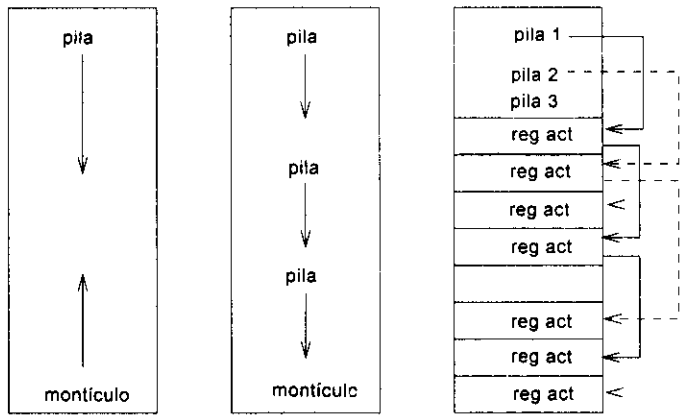


Figura 9.4. Registros de activación en un ambiente de tarea.

mutua que se describen más adelante. Cada una de las tareas requiere su propia gestión de almacenamiento, comúnmente una pila. Puesto que cada tarea se ejecuta en forma independiente, esto plantea al implementador un problema difícil: ¿cómo implementar pilas múltiples en una sola computadora?

En la figura 9.4 se describen varias soluciones a este problema. La figura 9.4(a) representa el enfoque usual para tareas individuales según se representan en lenguajes como Pascal o C, donde la pila y el montículo se crean en extremos opuestos del almacenamiento principal y, si llegan a encontrarse, entonces ya no hay más espacio disponible y el programa debe terminar. Todo el espacio se usa de manera eficiente, aunque, como se describió en la sección 5.4.6, cierto espacio puede no ser utilizable debido a fragmentación de la memoria en el montículo.

Si hay memoria suficiente se puede usar la figura 9.4(b). Cada tarea tiene su propia pila en un lugar aparte en la memoria y, otra vez como en el caso anterior, si cualquier pila se traslapa al segmento siguiente de memoria, el programa debe terminar. Con los sistemas modernos de memoria virtual, ésta es una solución eficaz. Por ejemplo, un programa puede tener un espacio de direcciones de mil millones de ubicaciones, pero sólo los pocos miles que se usan efectivamente están en la memoria real. Si la pila de cada tarea se inicia lo suficientemente apartada (por ejemplo, $pila_1$ en la localidad 100 millones, $pila_2$ en 200 millones, $pila_2$ en 300 millones, *montículo* en 400 millones), entonces es poco probable que se traslapen, y la gestión de almacenamiento del sistema operativo subyacente se puede usar para administrar estas pilas múltiples sin que el traductor del lenguaje requiera mucha ayuda adicional, aparte de establecer la dirección inicial de la pila.

Para sistemas que sólo tienen memoria limitada (es decir, no hay un sistema de memoria virtual o el espacio de direcciones de memoria virtual es relativamente pequeño), la opción anterior puede no ser realista. Se puede usar la figura 9.4(c). En este caso, toda la memoria es un montículo, y cada pila se compone de registros de activación asignados desde el montículo

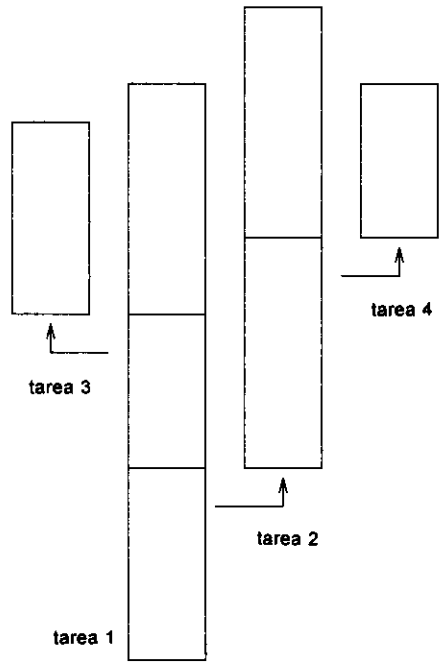


Figura 9.5. Modelo de pila cactus de tareas múltiples.

vinculados entre sí. Esto siempre funciona y era, por ejemplo, el mecanismo de almacenamiento para los primeros compiladores de PL/I. Sin embargo, ésta es una opción con elevados costos de estructura. Todas las activaciones y regresos de procedimientos requieren llamadas al asignador de memoria del sistema, las cuales llevan mucho tiempo, en vez de usar el eficiente modelo con base en pilas. Si cada registro de activación es de un tamaño particular, la fragmentación de la memoria, que no ocurre en el modelo de pilas, se convierte en un problema crítico. Las comparaciones de tiempos que se hacen empleando este modelo suelen mostrar que en esencia todo el tiempo de ejecución se gasta en las rutinas de asignación de memoria del sistema, y es poco lo que el programador puede hacer para acelerar estos programas. Esta tercera opción no es deseable, pero no suele haber muchas opciones cuando es necesario implementar tareas que se ejecutan en forma independiente utilizando un espacio reducido de direcciones de memoria.

Puesto que una tarea requiere su propia área central de pila para guardar los registros de activación de los subprogramas que llama, la ejecución se inicia con una sola pila para el procedimiento principal. Conforme se inician tareas, cada una requiere una nueva área de pila. La pila original se ha dividido, por tanto, en varias pilas nuevas, las cuales se pueden dividir otra vez si se inician más tareas. Esta estructura de gestión de almacenamiento se conoce comúnmente como una *pila de cactus*, puesto que se parece a la bifurcación de los brazos del saguaro, un cacto (figura 9.5). La conexión de la pila de la tarea recién iniciada con la pila de la unidad de programa dentro de la cual está anidada estáticamente se debe

mantener durante la ejecución debido a la posibilidad de que la tarea pueda hacer referencias no locales a datos compartidos en la pila de la unidad de programa original. Por tanto, se debe mantener un vínculo, como un apuntador de cadena estática, para permitir que se satisfagan correctamente estas referencias no locales durante la ejecución.

Exclusión mutua. Si la tarea *A* y la tarea *B* tienen acceso cada una a un solo objeto de datos *X*, entonces *A* y *B* deben sincronizar su acceso a *X*, de modo que *A* no esté en el proceso de asignar un nuevo valor a *X* mientras la tarea *B* está haciendo referencia simultáneamente a ese valor o asignando un valor distinto. Por ejemplo, si la variable *X* tiene el valor 1 y la tarea *A* ejecuta el enunciado:

```
if X > 0 then X := X + 1;
```

y *B* ejecuta el enunciado:

```
if X > 0 then X := X – 2;
```

entonces *X* puede acabar con el valor 0 (si *A* va en primer lugar), el valor –1 (si *B* va en primer lugar) o posiblemente el valor 2 (si ocurre que *A* y *B* intercalan sus acciones intentando ejecutar ambos enunciados en forma concurrente). Para asegurar que las dos tareas no intenten tener acceso y actualizar simultáneamente un objeto de datos compartido, una tarea debe ser capaz de obtener *acceso exclusivo* al objeto de datos mientras lo manipula. Existen varios enfoques distintos para resolver el problema de exclusión mutua cuando las tareas trabajan con datos compartidos. Ya hemos descrito los semáforos y la atomicidad en la sección 9.2, los cuales se pueden usar para sincronización de datos. En seguida se describen otros mecanismos.

Regiones críticas

Una *región crítica* es una serie de enunciados de programa dentro de una tarea, donde la tarea está operando sobre cierto objeto de datos que comparte con otras tareas. Si una región crítica de la tarea *A* está manipulando el objeto de datos *X*, entonces la exclusión mutua requiere que ninguna otra tarea esté ejecutando simultáneamente una región crítica que también manipule a *X*. Durante la ejecución de la tarea *A*, cuando está por comenzar la ejecución de la región crítica, *A* debe esperar hasta que cualquier otra tarea haya completado una región crítica que manipule a *X*. Cuando la tarea *A* inicia su región crítica, todas las demás tareas deben quedar excluidas para que no puedan entrar en sus regiones críticas (para la variable *X*) hasta que *A* haya completado su región crítica. Las regiones críticas se pueden implementar en las tareas asociando un semáforo con cada objeto de datos (o grupo de objetos) compartido. Los objetos de datos compartidos se hacen parte ordinariamente de un ambiente común explícito (o varios ambientes comunes) que es accesible para cada tarea.

Monitores

Otro enfoque para la exclusión mutua es a través del uso de un monitor. Un *monitor* es un objeto de datos compartido junto con el conjunto de operaciones capaces de manipularlo. Por

tanto, un monitor es similar a un objeto de datos definido por un tipo de datos abstracto, como se describe en la sección 5.1. Una tarea sólo puede manipular el objeto de datos compartido usando las operaciones definidas, de modo que el objeto de datos esté encapsulado, como es usual para objetos de datos definidos por tipos de datos abstractos. Para hacer valer la exclusión mutua, sólo es necesario exigir que *cuando mucho una* de las operaciones definidas para el objeto de datos pueda estar en ejecución en cualquier momento dado.

El requisito de exclusión mutua y encapsulamiento en un monitor hace que resulte natural representar el monitor mismo como una tarea. El objeto de datos compartido se convierte en un objeto de datos local dentro de la tarea, y las operaciones se definen como subprogramas locales dentro de la tarea. Por ejemplo, supóngase que el objeto de datos compartido es una tabla, *TablaGrande*, y se definen dos operaciones, *IntroducirElementoNuevo* y *EncontrarElemento*. La exclusión mutua es necesaria para que una tarea no esté introduciendo un nuevo elemento al mismo tiempo que otra tarea está intentando encontrar un elemento en la misma posición de la tabla. En Ada, el monitor se podría representar como una tarea *GestorDeTabla*, con dos entradas, *IntroducirElementoNuevo* y *EncontrarElemento*, definidas como se muestra en la figura 9.6. Dentro de *GestorDeTabla*, *TablaGrande* es una variable local. Se usa un enunciado **select** con dos alternativas **accept** para permitir que el monitor responda a las solicitudes de otras tareas en cuanto a efectuar una u otra de las operaciones *IntroducirElementoNuevo* y *EncontrarElemento* de tal manera que nunca esté en ejecución más de una de esas operaciones a la vez. Dos tareas pueden solicitar simultáneamente que se introduzcan o busquen elementos en la tabla. Por ejemplo, la tarea *A* puede ejecutar el enunciado de llamada de entrada:

```
IntroducirElementoNuevo(...);
```

y la tarea *B* puede ejecutar en forma concurrente la llamada de entrada:

```
EncontrarElemento( ... );
```

Se procesa la primera llamada de entrada que recibe *GestorDeTabla* (es decir, el encuentro tiene lugar como se describe en la sección 9.2). Si el segundo enunciado de entrada se ejecuta antes de que *GestorDeTabla* procese el primero, la segunda tarea deberá esperar. En esta forma, *TablaGrande* está protegido contra acceso simultáneo por parte de dos tareas.

Paso de mensajes

Otra solución al problema de datos compartidos entre tareas consiste en prohibir *objetos* de datos compartidos y proporcionar sólo la compartición de *valores* de datos a través del paso de valores como mensajes. Se trata simplemente del concepto de mensaje expuesto antes para la sincronización de datos. El uso del paso de mensajes como base para compartir datos asegura la exclusión mutua sin mecanismos especiales, puesto que cada objeto de datos es propiedad de exactamente una sola tarea, y ninguna otra tarea puede tener acceso directo al objeto de datos. La tarea propietaria *A* puede enviar una copia de los valores guardados en el objeto de datos *B* para su procesamiento. *B* tiene entonces su propia copia local del objeto de datos. Cuando *B* ha terminado de procesar su copia local, envía una copia de los nuevos valores a *A*, y *A* modifica entonces el objeto de datos real. La tarea *A*, desde luego, puede continuar modificando el objeto de datos real mientras *B* modifica su copia local.

```
task GestorDeTabla is
   entry IntroducirElementoNuevo(...);
   entry EncontrarElemento(...);
end;
task body GestorDeTabla is
   TablaGrande: array (...) of
   procedure Introducir(...) is
      –Enunciados para introducir elemento en TablaGrande
   end Introducir;
   function Encontrar(...) returns ... is
      – Enunciados para encontrar elemento en TablaGrande
   end Encontrar;
begin
      –Enunciados para inicializar Tabla Grande
   loop – Describir iteración siempre para procesar solicitudes de entrada
      select
         accept IntroducirElementoNuevo(...) do
            –Llamar Introducir para introducir elemento recibido en TablaGrande
         end;
      or accept EncontrarElemento(...) do
            – Llamar Encontrar para buscar elemento recibido en TablaGrande
         end;
      end select;
   end loop;
end GestorDeTabla;
```

Figura 9.6. Monitor representado como una tarea en Ada.

9.3 PROPIEDADES FORMALES DE LOS LENGUAJES

Gran parte del diseño e implementación de los primeros lenguajes tenía como base preocupaciones prácticas: ¿qué podemos hacer para que esta primitiva pieza de hardware de computadora sirva para construir los grandes programas que se necesitan para problemas prácticos como el diseño de aviones, el análisis de datos de radar o la vigilancia de reacciones nucleares? Muchos diseños tempranos de lenguajes fueron producto de esta clase de necesidades, con pocas aportaciones teóricas. Los defectos, fallas y éxitos de esta práctica condujeron a algunos modelos formales tempranos de sintaxis y semántica de lenguajes de programación, lo que a su vez llevó a mejores lenguajes. A pesar de ser mejores, estos lenguajes tenían todavía numerosos defectos tanto de diseño como de implementación. Mejores modelos teóricos condujeron a nuevos refinamientos de diseño.

Un modelo teórico puede ser *conceptual* (es decir, cualitativo) y describir la práctica en términos de un conjunto subyacente de conceptos básicos, sin que intente suministrar una descripción matemática formal de los conceptos. En este sentido, los capítulos precedentes han construido un modelo teórico de los conceptos básicos que sustentan el diseño e implementación de lenguajes de programación. Alternativamente, un modelo teórico puede ser *formal* (es decir, cuantitativo) y describir la práctica en términos de un modelo matemático preciso que se puede estudiar, analizar

y manipular usando las herramientas de las matemáticas. Los modelos teóricos de esta sección son modelos formales. El tratamiento presente no pretende ser completo, pero deberá proporcionar detalles suficientes para que el lector entienda el problema y la solución que se propone.

La sintaxis de lenguajes fue uno de los primeros modelos formales que se aplicó al diseño de lenguajes de programación. En esta sección se complementa el tratamiento que se hizo antes en el capítulo 3.

9.3.1 Jerarquía de Chomsky

Como se describió en el capítulo 3, las gramáticas BNF han probado ser muy útiles para describir la sintaxis de los lenguajes de programación. Sin embargo, estas gramáticas BNF, o libres del contexto, son sólo un componente de una clase de gramáticas descritas por Noam Chomsky en 1959 [CHOMSKY 1959]. Se bosqueja en forma breve el modelo de gramáticas que Chomsky definió inicialmente.

Tipos de gramáticas

En la sección 3.3.1 se proporcionó la sintaxis básica para reglas o *producciones* BNF. Repasaremos estas reglas y las generalizaremos para incluir una clase más amplia de gramáticas. Una *gramática* se define como un conjunto de símbolos no terminales, un conjunto de símbolos terminales, un símbolo de partida (uno de los terminales) y un conjunto de producciones. Cada clase de gramática se distingue por el conjunto de reglas de producción que se permiten.

Como ya se ha indicado en el caso de BNF, un *lenguaje* es simplemente un conjunto de series finitas de símbolos tomados de cierto alfabeto arbitrario que se deriva del símbolo de partida. El alfabeto es sólo el conjunto de caracteres que se pueden usar en programas, y cada serie finita de estos caracteres representa un programa válido completo. Se puede hablar acerca del conjunto de cadenas que una gramática *genera* (por ejemplo, las cadenas de terminales producidas a partir del símbolo de partida) o, de manera opcional, puede decirse que la gramática *reconoce* las cadenas (por ejemplo, a partir de una cadena, se puede producir su árbol sintáctico en retroceso hasta el símbolo de partida).

Un lenguaje de tipo n es aquel que es generado por una gramática de tipo n, donde no existe una gramática de tipo $n + 1$ que también la genere. (Como se verá, toda gramática de tipo n es, por definición, también una gramática de tipo $n - 1$).

Las gramáticas de tipo 3 son simplemente las gramáticas normales que definen los lenguajes de estados finitos que sirven de modelo para los componentes léxicos de un lenguaje. Las gramáticas de tipo 2 son nuestras familiares gramáticas BNF. Las gramáticas de tipo 1 y tipo 0 sirven para pocos fines prácticos en cuestiones de lenguajes de programación, pero son importantes en las áreas teóricas de las ciencias de la computación. En las subsecciones que siguen se sintetiza cada uno de los tipos.

Regulardores de gramática — Tipo 3

Como ya se ha señalado los autómatas de estados finitos y las gramáticas normales suministran un modelo para construir analizadores léxicos en un traductor para un lenguaje de programación (sección 3.3.2).

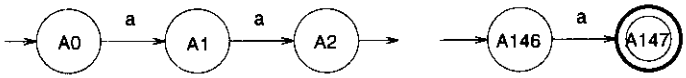


Figura 9.7. Uso de un FSA (AEF) para contar.

Las gramáticas normales tienen las propiedades siguientes:

- Casi todas las propiedades de estas gramáticas son *resolubles* (por ejemplo, ¿genera cadenas la gramática? ¿Genera la gramática una cadena dada del lenguaje? ¿Existe un número finito de cadenas en el lenguaje?)

- Las gramáticas normales pueden generar cadenas de la forma α^n para cualquier serie finita α y cualquier entero n. Es decir, la gramática puede reconocer cualquier número de patrones de longitud finita.

- Las gramáticas normales pueden contar hasta cualquier número finito. Por ejemplo, se puede reconocer $\{a^n \mid n = 147\}$ construyendo un FSA (AEF) con al menos 148 estados, como se indica en la figura 9.7. Fallará para las primeras 146 entradas, pero aceptará la entrada número 147. Con sólo 148 estados, ningún FSA puede aceptar de manera confiable una entrada de cualquier número específico mayor de 147.

- Estas gramáticas se suelen emplear en analizadores léxicos de compiladores para reconocer palabras clave o componentes léxicos de un lenguaje ("if", "while", "begin", identificadores, números, cadenas).

Como ejemplo, una gramática normal para generar identificadores en Pascal (una letra seguida de cualquier número de letras o dígitos, es decir, *letra*{*letra* ∨ *dígito*}*) está dada por:

Ident→ **a**X| ... | **z**X| **a**| ... | **z**
X→ **a**X| ... | **z**X| 0 X| ... | 9 X| **a**| ... | **z**| 0 | ... | 9

Gramáticas libres de contexto — Tipo 2

Se trata de las reglas de producción BNF con las que el lector ya está familiarizado. Las producciones son de la forma:

$$\mathbf{X} \rightarrow \alpha$$

donde α es cualquier serie de símbolos terminales y no terminales.

Estas gramáticas tienen las propiedades siguientes:

- Muchas propiedades de estas gramáticas son resolubles (por ejemplo, ¿genera cadenas la gramática? ¿Genera la gramática una cadena dada del lenguaje? ¿Está vacío el lenguaje?

- Estas gramáticas se pueden usar para contar dos elementos y compararlos. Es decir, se caracterizan por cadenas como $a^n c b^n$ para cualquier n.

- Las gramáticas libres de contexto se pueden "implementar" a través de pilas. En cuanto al ejemplo anterior, para reconocer $a^n c b^n$ se puede apilar a^n, pasar por alto la c y luego comparar la pila con b^n para comparar las dos cadenas.

- Estas gramáticas se pueden usar para desarrollar en forma automática árboles sintácticos de programas, como se describe en el capítulo 3.

- En su mayor parte, las gramáticas de tipo 2 y tipo 3 ya no son temas de investigación interesantes. Todas las propiedades importantes parecen estar resueltas.

Como ejemplo, la gramática usual de expresiones está dada por:

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{T} \mid \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{T} * \mathbf{P} \mid \mathbf{P}$$
$$\mathbf{P} \rightarrow \mathbf{i} \mid (\mathbf{E})$$

Gramáticas sensibles al contexto — Tipo 1

Estas gramáticas se caracterizan por producciones de la forma:

$$\alpha \rightarrow \beta$$

donde α es cualquier cadena de no terminales, β es cualquier cadena de terminales o no terminales, y el número de símbolos en α es menor o igual al número de símbolos en β.

Algunas de las propiedades de las gramáticas sensibles al contexto son:

- Todas las cadenas del símbolo de partida tienen longitud *no decreciente*, puesto que todas las producciones deben conservar igual la longitud de la cadena o aumentarla.

- Las gramáticas sensibles al contexto generan cadenas que necesitan una cantidad fija de "memoria". Por ejemplo, pueden reconocer $a^n b^n c^n$, que no es reconocible para una gramática libre de contexto.

- Las gramáticas sensibles al contexto son por lo general demasiado complejas para ser de utilidad en aplicaciones de lenguajes de programación.

- Se desconocen ciertas propiedades de las gramáticas sensibles al contexto. En la sección 9.3.3 se bosqueja en forma breve el problema de demostrar la completez de NF, o si el no determinismo con sensibilidad al contexto es lo mismo que determinismo.

Como ejemplo, la gramática siguiente genera $xa^n b^n c^n$:

X → ABCX|Y
CB → BC
CA → AC
BA → AB
CCY → CYc
BCY → BYc
BBY → BYb
ABY → AYb
AAY → AYa
AY → xa

Gramáticas sin restricciones — Tipo 0

Estas gramáticas se caracterizan por producciones no restringidas de la forma: $\alpha \rightarrow \beta$, donde α es cualquier cadena de no terminales y β es cualquier cadena de terminales y no terminales.

Estas gramáticas tienen las propiedades siguientes:

- Se pueden usar para reconocer cualquier función computable. Por ejemplo, se puede dar (aunque no fácilmente) una gramática para la cadena $a^n b^{f(n)}$ que representa la función f. Dadas n a's, la gramática genera $f(n)$ b's.
- Casi todas las propiedades son *irresolubles* (sección 9.3.3). Esto es, no existe un proceso por el cual se pueda determinar si una propiedad dada es cierta para todas las gramáticas de tipo 0 (por ejemplo, ¿está vacío el lenguaje?). Esto difiere del caso sensible al contexto, donde muchas propiedades simplemente se desconocen y pueden ser ciertas, falsas o irresolubles.

9.3.2 Irresolubilidad

Al analizar la jerarquía de Chomsky, se podía ver que al pasar a gramáticas de tipo 3 a tipo 2, a tipo 1 y a tipo 0, los lenguajes resultantes se hacían más y más complejos. Sabemos que las computadoras actuales son extremadamente rápidas y a menudo parecen tener propiedades mágicas de solución. Así que podría ser deseable considerar esta cuestión: ¿existe un límite en cuanto a lo que se puede computar con una computadora?

Considérese el problema práctico siguiente. En vez de poner a prueba un programa en C, ¿se podría escribir otro programa que lea una descripción del programa en C (es decir, su archivo fuente) y determine si el programa en C se va a detener cuando se ejecute? Si se tuviera un programa así, sería extremadamente útil; se podrían evitar muchas pruebas de programas que finalmente entran en iteraciones infinitas verificándolos primero con este nuevo programa de prueba.

Si se intenta escribir este programa, se encontrará que es sumamente difícil; imposible, incluso. El problema no es que uno no sea lo suficientemente hábil; se trata de las limitaciones

de nuestro sistema matemático que un estudio de los lenguajes de tipo 0 pone de manifiesto. En esta sección se analizan algunos de los aspectos de este problema.

Una computadora universal sencilla

Cuando se programa en un lenguaje, digamos el lenguaje A, por lo común es claro que se podría escribir un programa equivalente en el lenguaje B. Por ejemplo, si se escribe un programa de cómputo de nóminas en COBOL, el mismo programa se podría escribir de manera equivalente en C o FORTRAN, y quizá con más dificultad en ML o LISP. ¿Existe un programa que se pueda escribir en un lenguaje, para el cual no hay realmente un programa equivalente posible en uno de los otros lenguajes? Por ejemplo, existe un programa que se pueda escribir en LISP o PROLOG y que no tenga un equivalente en FORTRAN? Un *lenguaje de programación universal* es un lenguaje que es lo suficientemente general para permitir la expresión de cualquier cómputo. El problema se podría expresar preguntando: *¿Son universales todos los lenguajes de programación normales?* Si no lo son, entonces, ¿qué clases de programas puede expresar un lenguaje, que otro no puede? Si lo son, ¿por qué necesitamos entonces todos estos lenguajes distintos? Si todos son universales, entonces quizá deberíamos encontrar el lenguaje de programación más simple y prescindir por completo de los otros.

Adviértase en primer lugar que la pregunta se puede formular en términos de la función que computa un programa. Decir que un programa P escrito en el lenguaje A es equivalente a un programa Q escrito en el lenguaje B debe significar que los dos programas computan la misma función. Es decir, cada programa toma los mismos conjuntos de datos de entrada posibles y produce los mismos resultados en todos los casos. Un lenguaje de programación universal es aquel en el cual cualquier *función computable* se puede expresar como un programa. Una función es computable si se puede expresar como un programa en algún lenguaje de programación. Este planteamiento del problema parece un poco tortuoso, porque nos enfrentamos a una dificultad fundamental: cómo definir el concepto de una función computable. Decir que una función es computable significa, intuitivamente, que existe algún procedimiento que se puede seguir para computarla, paso a paso, de tal manera que el procedimiento siempre termina. Pero para definir la clase de todas las funciones computables, es necesario proporcionar un lenguaje de programación o una computadora virtual que sea universal, donde se pueda expresar cualquier función computable. Pero no sabemos como distinguir si un lenguaje es universal; ése es el problema.

Este problema ya se había considerado incluso antes de la invención de las computadoras. Matemáticos de la década de 1930 habían estudiado el problema de definir la clase de funciones computables. Varios de ellos habían inventado máquinas abstractas sencillas o autómatas que podrían servir para definir la clase de funciones computables. La más ampliamente conocida de ellas es la *máquina de Turing*, llamada así en honor de su inventor, Alan Turing [TURING 1936].

Una máquina de Turing tiene una única estructura de datos, un arreglo lineal de longitud variable llamado *cinta*. Cada componente de la cinta contiene sólo un carácter. También hay una sola variable apuntador, llamada *cabeza de lectura*, que apunta a cierto componente de la

cinta en todo momento. La máquina de Turing es controlada por un programa en el que intervienen sólo unas pocas operaciones sencillas:

1. El carácter de la posición de cinta que designa la cabeza de lectura puede ser leído o escrito (reemplazado por otro carácter). El programa se puede bifurcar de acuerdo con el valor leído. También puede usar enunciados **goto** no condicionales para construir iteraciones. (Es decir, la lógica interna de la máquina de Turing es similar a la del autómata de estados finitos dada previamente.)

2. El apuntador de cabeza de lectura se puede modificar de manera que señale al componente de la cinta una posición a la izquierda o una a la derecha de su posición actual. Si este desplazamiento de la cabeza de lectura lleva a su apuntador fuera del extremo de la cinta, entonces se inserta un nuevo componente en ese extremo, con el valor inicial del carácter nulo.

Cuando se inicia la operación de una máquina de Turing su cinta contiene los datos de entrada y su cabeza de lectura está situada en el carácter más a la izquierda de los datos de entrada. La máquina de Turing ejecuta una serie de las sencillas operaciones antes citadas y durante el proceso modifica el contenido de su cinta (y posiblemente también lo amplía). Si finalmente se detiene, la cinta contiene los resultados computados.

Una máquina de Turing es una máquina abstracta extremadamente sencilla. Si se desea agregarle dos números, se debe programar usando sólo las operaciones citadas. Y no se permiten otras variables o estructuras de datos; sólo la cinta única para almacenamiento (pero adviértase que la capacidad de almacenamiento de la cinta misma es ilimitada).

¿Puede una máquina de Turing hacer *algo* útil? Se podrían presentar varios programas de ejemplo para convencer al lector de que al menos se puede programar para hacer ciertas cosas sencillas como adición y sustracción. Sin embargo, lo que se desea argumentar es algo mucho más fuerte: una máquina de Turing puede hacer *todo* lo útil (¡en los dominios de la computación!). Es decir, nos gustaría demostrar que todo cómputo se puede expresar como un programa para una máquina de Turing y, por tanto, que el lenguaje que se usa para programar una máquina de Turing es un lenguaje universal, a pesar del hecho de que permite sólo un vector y nada de aritmética, ninguna subrutina y ninguna de las otras estructuras que se asocian con los lenguajes ordinarios de programación.

El enunciado formal de esta idea se conoce como la *tesis de Church* (el mismo Church que se menciona en la sección 9.4): *Cualquier función computable puede ser computada por una máquina de Turing*. La tesis de Church no es un teorema que se pueda probar. Es una hipótesis que se podría refutar si se pudiera definir un cómputo en otro lenguaje de programación y demostrar que no tiene un equivalente que sea un cómputo para una máquina de Turing. Sin embargo, la tesis de Church ha sido examinada durante varios años por muchos matemáticos. Se han inventado numerosas computadoras y lenguajes reales y abstractos, y se ha probado repetidamente que cada método alternativo propuesto para producir una máquina o lenguaje universal no es, de hecho, más poderoso que una máquina de Turing. Es decir, toda función que se puede computar usando el nuevo lenguaje o máquina se puede representar como un programa para una máquina de Turing.

Un resultado que viene al caso en nuestro análisis de los lenguajes de programación es que las máquinas de Turing son equivalentes a las gramáticas de tipo 0 que se estudiaron antes en este capítulo. Es bastante fácil demostrar que cualquier estado de una máquina de Turing se puede simular por medio de una deducción con una gramática de tipo 0. De manera similar, cualquier gramática de tipo 0 puede ser reconocida por una máquina de Turing (no determinista). A diferencia del caso libre de contexto, lo que es bastante sorprendente, las máquinas de Turing no deterministas y deterministas son equivalentes. Es decir, cualquier cómputo efectuado por una máquina de Turing no determinista puede ser realizado por una máquina de Turing determinista equivalente.

El hecho fundamental que la máquina de Turing ilustra es éste: *Casi no se requiere de un mecanismo para conseguir la universalidad, aparte de alguna clase de capacidad de almacenamiento ilimitada.* Incluso un conjunto extremadamente simple de estructuras de datos y operaciones es suficiente para permitir la expresión de cualquier función computable.

El problema de paro

El estudio de las máquinas de Turing y los lenguajes ha conducido a algunos otros resultados importantes. Entre estos hay varios que muestran que ciertos problemas son *irresolubles*. Es decir, no existe un algoritmo general para su solución, incluso en el contexto de estas máquinas sencillas.

Considérese el problema del programa para verificar si un programa dado en C se detendrá alguna vez, como se describió al principio de esta sección. Sin duda se sabe de subcasos de programas en C que se paran. Por ejemplo, el programa:

```
principal()
    {int i;
    i=0;
    }
```

ciertamente se detiene. De hecho, el conjunto de todos los programas en C compuestos de sólo declaraciones y enunciados de asignación se detiene. Incluso se puede construir fácilmente un programa que determina esto para nosotros, como sigue:

1. Tomar cierto compilador de C.
2. De la gramática BNF que define a C, eliminar todas las producciones que incluyen cualquier bifurcación, llamada de procedimiento o enunciados de iteración.
3. Ejecutar el compilador usando este conjunto reducido de producciones para C.

Cualquier programa correctamente compilado por este compilador de C modificado se compone sólo de enunciados de asignación y se detendrá. Los casos interesantes son aquellos programas que no compilan en este compilador de C modificado. Algunos son obviamente correctos y se detendrán (por ejemplo, todos los programas que el lector ha escrito y que contienen iteraciones que trabajan correctamente), pero otros pueden no parar y entonces describirán ciclos indefinidamente. El problema es que no se puede saber en general cuál es el caso que se tiene.

Supóngase que se tiene un programa verificador universal de este tipo y que se le proporciona un programa extremadamente largo para analizar. Supóngase también que se le deja en ejecución hasta por tres años, y si en ese tiempo no concluye que el programa se detiene, asúmase que nunca lo hará.

Si se le deja en ejecución por tres años y luego se interrumpe y se afirma que el programa que está analizando nunca se detendrá, quizá la respuesta se habría podido encontrar si se hubiera ejecutado el programa por 3 años y 10 minutos. El problema es que, para elementos que no están en el conjunto computable, simplemente no se sabe cuándo dejar de intentar el cómputo de la respuesta.

Esto es un análisis intuitivo de lo que se ha venido a conocer como el *problema de paro*: ¿existe un algoritmo general para determinar si cualquier máquina de Turing dada se detendrá alguna vez cuando se le proporciona cualquier cadena particular de caracteres como sus datos de entrada? Los estudios teóricos originales de Turing en 1936 demostraron que el problema de paro era irresoluble: no podía existir un algoritmo general para resolver el problema para todas las máquinas de Turing y todos los conjuntos de datos de entrada.

Para demostrar que cualquier problema particular es irresoluble, se expone como equivalente al problema de paro. Si se pudiera resolver entonces este problema de interés, también se podría resolver el problema de paro, del cual se sabe que es irresoluble. Por consiguiente, el problema original también debe ser irresoluble.

El estudio de estos lenguajes universales y máquinas sencillas conduce a la conclusión de que cualquier lenguaje de programación que se pueda usar razonablemente en la práctica es sin duda un lenguaje universal, si no se toman en cuenta los límites de tiempo de ejecución y almacenamiento. Así, el programador que rehúsa resueltamente emplear cualquier lenguaje que no sea, por ejemplo, LISP, porque "se puede hacer cualquier cosa en LISP", está de hecho en lo correcto. Quizá sea difícil, pero sin duda se puede hacer. Las diferencias entre lenguajes de programación no son diferencias *cuantitativas* respecto a lo que se puede hacer, sino sólo diferencias *cualitativas* en cuanto a la elegancia, facilidad y eficacia con que se pueden hacer las cosas.

Ambigüedad

La jerarquía de Chomsky de formas de gramáticas es un ejemplo de un fenómeno común en el trabajo teórico: se pueden construir muchos modelos teóricos distintos de una situación práctica, y cada uno abstrae el problema práctico de diferentes maneras. Algunos de estos modelos teóricos son menos útiles que otros, y se estudian por un tiempo y luego se olvidan. Otros captan cierto elemento importante del problema práctico y entran al uso general. Las gramáticas sensibles al contexto y las no restringidas de la jerarquía de Chomsky resultaron ser los modelos equivocados para lenguajes de programación. A pesar de ser más poderosas que las BNF, también eran más difíciles de entender, analizar y usar en la práctica. Muchas de las formas de gramática posteriores se desarrollaron en un intento por superar las limitaciones de estos modelos.

Las gramáticas BNF tienen varias propiedades atractivas. Su uso es bastante fácil en situaciones prácticas y son lo suficientemente poderosas para expresar casi todas (pero no todas) las restricciones sintácticas que se requieren para los lenguajes de programación (los

Nivel de Chomsky	Clase de Gramática	Cláse de Máquina
0	Sin restricciones	Máquina de Turing
1	Sensible al contexto	Autómata linealmente acotado
2	Libre del contexto	Autómata de desplazamiento descendente
3	Normal	Autómata de estados finitos

Tabla 9.1. Clases de gramáticas y máquinas abstractas.

lingüistas las hallaron menos útiles para los lenguajes naturales). También es fácil analizarlas matemáticamente para descubrir hechos acerca de los lenguajes que ellas definen y que no son obvios de inmediato

Por ejemplo, una cuestión práctica importante es la de si una gramática BNF para un lenguaje de programación es *ambigua*; es decir, ¿define una gramática un análisis sintáctico único para cada programa válido, o son posibles múltiples análisis sintácticos? Ordinariamente, cada análisis sintáctico corresponde a un significado diferente que se da al programa, de modo que múltiples análisis sintácticos conducen a múltiples significados. Surge esta cuestión: ¿se puede encontrar un procedimiento general para determinar si una gramática BNF es ambigua? El resultado a partir de estudios teóricos es sorprendente: no hay que molestarse en tratar de encontrar un procedimiento así. No existe. En términos formales, se dice que la cuestión de determinar la ambigüedad es *irresoluble*; no puede haber un procedimiento general para responder la pregunta para cualquier gramática BNF. El resultado es decepcionante porque, ante una gramática BNF compleja que contiene cientos de producciones, sería oportuno contar con un programa que pudiera dar una respuesta de sí o no casi siempre, pero la irresolubilidad de la cuestión nos dice que ningún programa podría dar *siempre* una respuesta. Para ciertas gramáticas, tendría que ser incapaz de proporcionar una respuesta, no importa cuánto tiempo se deje en ejecución.

9.3.3 Complejidad de algoritmos

La jerarquía de Chomsky es todavía un tema de estudio fascinante en la ciencia formal de la computación. Aunque las cuestiones de análisis léxico y análisis sintáctico están resueltas en gran medida, existen muchas preguntas interesantes que es necesario resolver. A continuación se sintetizan algunos de estos problemas restantes.

Gramáticas y máquinas. Para cada clase de gramática se dijo que había una dualidad entre esa clase de gramática y una clase de máquina abstracta. La tabla 9.1 y la figura 9.8 resumen este comentario:

1. Un autómata de estados finitos [figura 9.8(a)] se compone de una representación gráfica de estados finitos y una cinta de un solo sentido. Para cada operación, el autómata lee el símbolo siguiente de la cinta y entra en un nuevo estado.

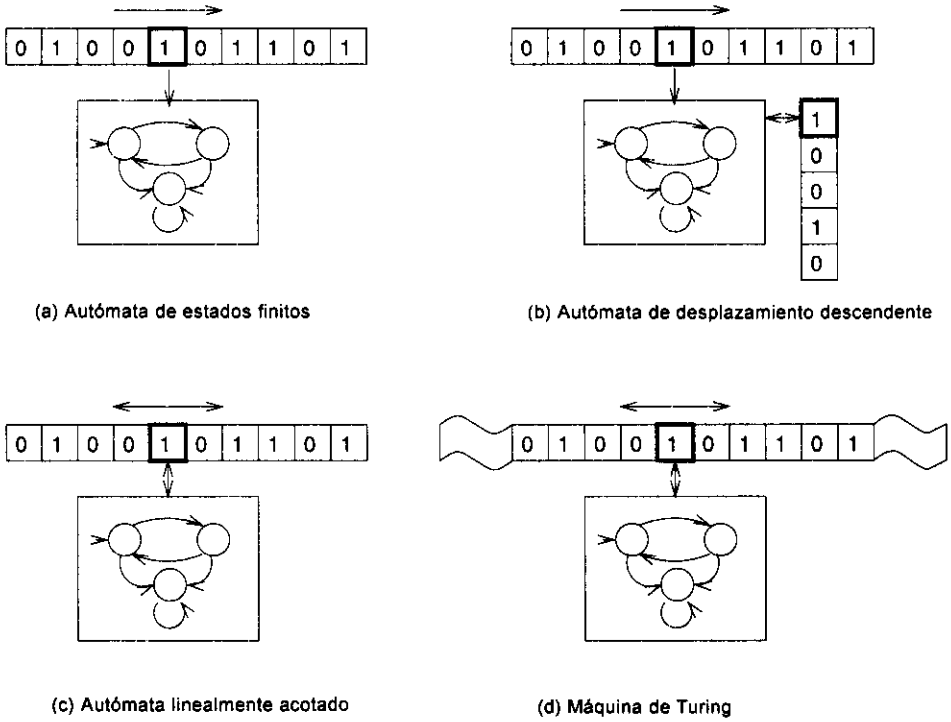


Figura 9.8. Modelos de máquinas abstractas.

2. Un autómata de desplazamiento descendente [figura 9.8(b)] agrega una pila al autómata finito. Para cada operación, el autómata lee el siguiente símbolo de la cinta y el símbolo de la pila, escribe un nuevo símbolo de pila y entra en un nuevo estado.
3. Un autómata linealmente acotado [figura 9.8(c)] es similar al autómata de estados finitos, con el agregado de que puede leer y escribir en la cinta para cada símbolo de entrada y puede mover la cinta en un sentido u otro.
4. Una máquina de Turing [figura 9.8(d)] es similar a un autómata linealmente acotado, excepto que la cinta es infinita en ambos sentidos.

Ya hemos analizado todos éstos anteriormente, excepto el autómata linealmente acotado. Se define un autómata linealmente acotado (LBA, *linear-bounded automaton*) como una máquina de Turing que puede usar sólo la cantidad de cinta que contiene sus datos de entrada. Por tanto, su almacenamiento crece con el tamaño de la entrada y puede reconocer cadenas más complejas. Sin embargo, sus capacidades de almacenamiento de datos son limitadas en comparación con la máquina de Turing, puesto que no puede alargar su cinta indefinidamente.

Cuando se estudia el poder de cómputo de la forma tanto determinista como no determinista de cada máquina abstracta, se tienen estos resultados interesantes:

Tipo de Máquina	Determinismo contra no determinismo
Autómata de estados finitos	Igual
Autómata de desplazamiento descendente	No igual
Autómata linealmente acotado	?
Máquina de Turing	Igual

Para lenguajes tanto de tipo 3 como de tipo 0, el no determinismo no agrega poder adicional a la máquina abstracta. La funcionalidad de la máquina no determinista puede ser reproducida por una máquina determinista equivalente. Por otra parte, para lenguajes de tipo 2, se sabe que el no determinismo aumenta el poder. Se puede reconocer el conjunto de palíndromos por medio de un autómata de desplazamiento descendente no determinista (como se describe en la sección 3.3.4) haciendo que la máquina no determinista adivine dónde está el punto medio de la cadena, pero no se puede reconocer esta cadena con un autómata de desplazamiento descendente determinista. El autómata de desplazamiento descendente determinista reconoce los *lenguajes deterministas libres de contexto*, que resultan ser los lenguajes LR(k), los cuales constituyen la base de la teoría del análisis sintáctico de compiladores.

Desafortunadamente, no se tiene una respuesta para el caso linealmente acotado. Simplemente no se sabe si los LBA no deterministas son equivalentes a los LBA deterministas. Este problema ha atormentado a los científicos teóricos de la computación por más de 30 años.

Cómputos en tiempo polinómico. A causa de la dificultad para resolver la cuestión del no determinismo para los LBA, ha surgido una especialidad completa en la ciencia teórica de la computación, la determinación de la complejidad de un algoritmo. El determinismo se puede relacionar con el cómputo en *tiempo polinómico*. Es decir, dada una entrada de longitud n, la máquina abstracta se ejecutará, cuando mucho, durante $p(n)$ unidades de tiempo para cierto polinomio p (donde $p(x) = a_1x^n + a_2x^{n-1} + \ldots + a_{n-1}x + a^n$). El no determinismo, por otra parte, se reduce demostrando que un cómputo no se puede completar en tiempo polinómico. Se ha demostrado que la equivalencia del LBA no determinista y el LBA determinista es lo mismo que el problema siguiente: dado un cómputo que se puede completar en tiempo no polinómico (NP), ¿existe otra máquina capaz de computar el mismo resultado en tiempo polinómico (P)? O, en otras palabras, ¿es P=NP? A este problema para LBA le llamamos *completez NP* y decimos que cualquier problema equivalente a esta pregunta es *completo NP*. La cuestión del no determinismo para LBA es por tanto completa NP.

Existen muchos problemas de los que se ha demostrado que son completos NP, pero hasta ahora no se tiene una respuesta para la pregunta básica. Muchos teóricos piensan actualmente que no existe una respuesta para esta pregunta. Simplemente está fuera del sistema de axiomas de nuestro formalismo matemático.

Valor de los modelos formales

Los programadores en ejercicio de su profesión no suelen apreciar el valor de los modelos y estudios teóricos en el progreso de los lenguajes y técnicas de programación. La abstracción que es necesaria como primer paso en la construcción de un modelo teórico para un problema

práctico a veces parece simplificarse hasta el punto de que se pierde el meollo del problema práctico. De hecho, esto puede y suele ocurrir; si elige la abstracción equivocada, el teórico puede estudiar un modelo teórico y producir resultados que no se pueden traducir en una solución al problema práctico original. Sin embargo, como esta sección ha tratado de indicar, cuando se encuentra la abstracción correcta, los estudios teóricos pueden producir resultados del más profundo impacto práctico.

9.4 SEMÁNTICA DE LENGUAJES

Aunque es mucho lo que se sabe acerca de la sintaxis de los lenguajes de programación, se conoce en menor grado cómo definir correctamente la semántica de un lenguaje. En la sección 3.3.5 se han analizado las gramáticas de atributos. Aquí se estudian otros modelos con mayor detalle.

9.4.1 Semántica denotativa

La semántica denotativa es un modelo aplicativo formal para describir la semántica de lenguajes de programación. Antes de analizar en forma breve la semántica denotativa, primero se presentará el cálculo lambda como un modelo funcional más sencillo desarrollado durante los años treinta como una forma de explicar los cómputos matemáticos. A partir del cálculo lambda, se pueden desarrollar estructuras más complejas que incluyen los conceptos de tipos de datos y semántica de lenguajes de programación.

Cálculo lambda

Quizá el primer modelo de semántica de lenguajes de programación fue el *cálculo lambda* (cálculo λ), desarrollado en los años treinta por A. Church como un modelo teórico de la computación comparable con la máquina de Turing (sección 9.3.3). Aunque antecedió a las primeras computadoras en varios años y a los lenguajes de programación por alrededor de 15 años, el cálculo λ sirvió como un buen modelo para la invocación de funciones de lenguajes de programación. De hecho, tanto ALGOL como LISP pueden seguir el rastro de su semántica de llamadas de función hasta el modelo del cálculo λ; la sustitución del cálculo λ es una correspondencia directa del mecanismo de paso de parámetros de llamada por nombre definido en ALGOL (sección 7.3.1). El cálculo λ fue ampliado por Scott [SCOTT 1972] a una teoría general de tipos de datos, la cual se conoce en la actualidad como *semántica denotativa*. Esta notación tuvo un impacto sobre el diseño de la teoría de tipos de datos del ML. Un estudio completo tanto del cálculo l como de la semántica denotativa está fuera del alcance de este libro; sin embargo, se presenta una breve perspectiva de estas ideas y se muestra su relación con el diseño de lenguajes de programación.

Una expresión se define de manera recursiva como:

1. Si x es un nombre de variable, entonces x es una expresión λ.
2. Si M es una expresión λ, entonces $\lambda x.M$ es una expresión λ.
3. Si F y A son expresiones λ, entonces $(F\ A)$ es una expresión λ. F es el *operador* y A es el *operando*.

También podemos definir expresiones λ por medio de una gramática libre de contexto:

expr_λ → *identificador* | λ*identificador*λ.*expr_λ* | (*expr_λ expr_λ*)

Los siguientes son ejemplos de expresiones λ generadas por la gramática precedente:

x	$\lambda x.x$	λx.y
$\lambda x.(xy)$	$(\lambda x.(xx)\lambda x.(xx))$	$\lambda x.\lambda y.x$

Las variables pueden ser enlazadas o libres. Intuitivamente, una variable enlazada es una variable declarada de manera local, en tanto que una variable libre carece de declaración. En la expresión λ *x*, *x* es *libre*. Si *x* es libre en *M*, entonces *x* está *enlazada* en λ*x.M*. *x* es libre en (*F A*) si *x* es libre en *F* o *x* es libre en *A*.

Cualquier variable enlazada puede cambiar de nombre, de igual manera que en la reasignación de nombre a parámetros de función. Así, la expresión λ λ*x.x* es equivalente a λ*y.y*. λ*x.*λ*x.x* es equivalente a λ*x.*λ*y.y*, puesto que la variable *x* está enlazada a la λ*x* de la derecha. De manera informal, las variables enlazadas son "parámetros" para la función que describe la expresión λ; las variables libres son globales. Esta analogía muestra que la expresión λ es una apro-ximación simple al concepto de procedimiento o subrutina en casi todos los lenguajes de programación algorítmicos, como Pascal, C, Ada o Fortran.

Operaciones sobre expresiones lambda

Las expresiones λ tienen una sola operación de *reducción*. Si (*F A*) es una expresión λ y *F* = λ*x.M*, entonces todas las incidencias de *x* libre en *M* se pueden sustituir por *A*. Esto se escribe como: (λ*x.M A*) ⇒ *M'*. Esta operación es análoga a la sustitución de un parámetro formal por un argumento en una llamada de función.

Los siguientes son algunos ejemplos de reducciones de expresiones λ:

$(\lambda x.x\ y)$	$\Rightarrow y$	
$(\lambda x.(xy)\ y)$	$\Rightarrow (yy)$	
$(\lambda x.(xy)\ \lambda x.x)$	$\Rightarrow (\lambda x.x\ y)$	$\Rightarrow y$
$(\lambda x.(xx)\ \lambda x.(xx))$	$\Rightarrow (\lambda x.(xx)\ \lambda x.(xx))$	$\Rightarrow \ldots$

Adviértase que la operación de reducción no siempre da por resultado una expresión λ más simple que la expresión original; el cuarto de estos ejemplos no termina. Esto nos conduce a la *propiedad Church-Rosser*; si dos reducciones diferentes de una expresión λ terminan, entonces son miembros de la misma clase de valores. O, dicho de otra manera, si la expresión λ *M* tiene las reducciones *M* Þ *P* y *M* Þ *Q*, entonces existe una expresión λ única *R* tal que *P* Þ *R* y *Q* Þ *R*. Llamamos a *R* la *forma normal* de la clase de valores representada por *M*.

Paso de parámetros con expresiones λ. En cualquier expresión λ, considérense dos enfoques normales hacia la reducción: (1) reducir primero el término izquierdo más interno, y (2) reducir primero el término izquierdo más externo. Por ejemplo, en las expresiones λ: $(\lambda y.(yy)\ (\lambda x.(xx)\ a))$, el término más externo es la expresión completa, en tanto que el término más interno es $(\lambda x.(xx)\ a)$. Son posibles dos secuencias de reducción:

Más externo primero	**Más interno primero**
$(\lambda y.(yy)\ (\lambda x.(xx)\ a))$	$(\lambda y.(yy)\ (\lambda x.(xx)\ a))$
$\Rightarrow ((\lambda x.(xx)\ a)\ (\lambda x.(xx)\ a))$	$\Rightarrow (\text{l}y.(yy)\ (aa))$
$\Rightarrow ((aa)\ (aa))$	$\Rightarrow ((aa)\ (aa))$

Ambos reciben la misma respuesta; sin embargo, la reducción del más externo sustituyó el parámetro *y* por la función $(\lambda x.(xx)\ a)$ y evaluó esta función para cada incidencia de *y*. Esto es precisamente el mecanismo de llamada por nombre descrito en la sección 7.3.1. Por otra parte, la reducción del más interno evaluó primero la constante (aa) antes de hacer la sustitución de *y*. Esto es simplemente el mecanismo de llamada por valor.

Si existe una forma normal, la evaluación de llamada por nombre la encontrará, aunque, como se acaba de demostrar, ninguno de los métodos de reducción tiene que terminar. La llamada por nombre es una forma de evaluación perezosa (sección 6.2.2), de modo que sólo es necesario evaluar las expresiones que se usan en la solución final. La llamada por valor, sin embargo, siempre evalúa argumentos, por lo cual se evaluarán argumentos que no terminan incluso si no son necesarios.

Se puede usar esto para crear ejemplos sencillos de términos que finalizan a través de llamada por nombre y no terminan a través de llamada por valor. Todo lo que hay que hacer es elaborar una expresión λ que no termine, como un argumento al que nunca se hace referencia. Ya sabemos que $(\lambda x.(xx)\ (\lambda x.(xx))$ no termina y, de manera trivial en $\lambda x.z$, nunca se hace referencia al parámetro *y*; por tanto:

$$(\lambda x.z\ (\lambda x.(xx)\ \lambda x.(xx)))$$

tiene la forma normal *z*, la cual se determina con facilidad si se emplea llamada por nombre, pero representa una expresión λ que no termina si se emplea llamada por valor.

Modelos matemáticos con expresiones lambda

El cálculo λ se desarrolló originalmente como un modelo lógico de computación. Podemos usar esta clase de expresiones para crear modelos de nuestra comprensión de la aritmética. En primer término, se usarán expresiones λ para elaborar un modelo del cálculo de predicados y luego, usando cálculo de predicados, podemos crear modelos de enteros.

Valores booleanos. Se tomarán expresiones λ como modelos de valores booleanos:
Cierto (*C*) se definirá como: $\lambda x.\lambda y.x$. (La interpretación de esta expresión puede quedar más clara si se afirma que *cierto* significa: de un par de valores, elíjase el primero. El ejemplo que sigue demuestra esto.)
Falso (*F*) se definirá como: $\lambda x.\lambda y.y$. (De un par, elíjase el segundo.)

Hemos definido estos objetos, así que las propiedades siguientes son ciertas:

$((T\ P)Q) \Rightarrow P$ es decir, $((T\ P)Q) \Rightarrow ((\lambda x.\lambda y.x\ P)Q) \Rightarrow (\lambda y.P\ Q) \Rightarrow P$
$((F\ P)Q) \Rightarrow Q$ es decir, $((F\ P)Q) \Rightarrow ((\lambda x.\lambda y.y\ P)Q) \Rightarrow (\lambda y.y\ Q) \Rightarrow Q$

Dadas las definiciones para estas constantes T y F, se pueden definir las funciones booleanas:

$$\begin{aligned} \textit{not}\ (no) &= \lambda x.((xF)C) \\ \textit{and}\ (y) &= \lambda x.\lambda y.((xy)F) \\ \textit{or}\ (o) &= \lambda x.\lambda y.((xC)y) \end{aligned}$$

Con estas definiciones, necesitamos demostrar que nuestra interpretación de ellas es congruente con nuestras reglas de lógica de predicados. Por ejemplo, *not*, cuando se aplica a C regresa F, y *not* aplicado a F regresa C:

$$(\textit{not}\ C) = (\lambda x.((x\ F)C)C) \Rightarrow ((C\ F)C) \Rightarrow F$$
$$(\textit{not}\ F) = (\lambda x.((x\ F)C)F) \Rightarrow ((F\ F)C) \Rightarrow C$$

Se puede demostrar que *and* y *or* tienen las propiedades lógicas usuales en forma análoga.

Enteros. Dadas las funciones booleanas, ahora podemos desarrollar los enteros:

$$\begin{aligned} 0 &= \lambda f.\lambda c.c \\ 1 &= \lambda f.\lambda c.(fc) \\ 2 &= \lambda f.\lambda c.(f(fc)) \\ 3 &= \lambda f.\lambda c.(f(f(fc))) \\ \ldots &\quad \ldots \end{aligned}$$

c es el elemento "cero", y f es la función "sucesora" (es decir, suma 1) aplicada con suficiente frecuencia al elemento c. A partir de estas definiciones, se pueden definir las operaciones aritméticas ordinarias:

El entero N se escribirá como la expresión $\lambda\ (N\ a)$, la cual es $\lambda c.(a...(a\ c)...)$. Aplicando reducción a $((N\ a)b)$, se obtiene $(a...(a\ b)...)$.

Considérese $((M\ a)((N\ a)b))$ aplicando la constante $((N\ a)b)$ a la expresión $\lambda\ (M\ a)$. La sustitución de c por $((N\ a)b)$ en $(M\ a)$ produce $(a...(a\ b)...)$, donde ahora hay $(M + N)$ a's en la lista. Acabamos de demostrar que podemos *sumar* expresiones λ:

$$[M + N] = \lambda a.\lambda b.((M\ a)((N\ a)b))$$

o el operador + se define como:

$$+ = \lambda M.\lambda N.\lambda a.\lambda b.((M\ a)((N\ a)b))$$

De manera similar, se puede demostrar también que:

$$\begin{aligned} &\textbf{Multiplicación:} \quad [M \times N] = \quad \lambda a.(M(N\ a)) \\ &\textbf{Exponenciación:} \quad [M^N] = \quad (N\ M) \end{aligned}$$

Continuando de esta manera se pueden desarrollar todas las funciones matemáticas computables. Sin embargo, apliquemos estas ideas a la semántica de lenguajes de programación.

Modelos de lenguajes de programación

Dado el análisis anterior sobre expresiones lambda, se puede ampliar el concepto y usar expresiones lambda para elaborar modelos de tipos de datos y, a partir de eso, se puede ampliar el modelo para incluir la semántica de lenguajes de programación.

Cuando la operación de reducción se aplica a una expresión, las expresiones λ se reducen ya sea a constantes o a otras expresiones λ. Por tanto, todas las expresiones λ son soluciones de la ecuación funcional:

$$expresión\ \lambda = constante\ + (expresión\ \lambda \Rightarrow expresión\lambda)$$

Sin extraviarse demasiado en los detalles de la semántica denotativa, se afirmará simplemente que un *tipo de datos* es una solución a esta ecuación. Adviértase, sin embargo, que hay un análogo directo entre este modelo formal de tipo de datos y las definiciones de tipos en ML:

```
datatype Milista = var of int |
                   elemlista of int * Milista;
```

el cual define una lista de enteros (constructor *Milista*), usando los constructores (como las expresiones λ) *var* y *elemlista*.

Usando este concepto se puede definir un modelo de *semántica denotativa* de un lenguaje de programación simple. El modelo es una forma de semántica de operación, puesto que estamos "rastreando" a través de la ejecución de cada tipo de enunciado para determinar su efecto sobre un intérprete de mayor nivel. Sin embargo, a diferencia de los intérpretes tradicionales, como los que se describen en el capítulo 2, consideramos que un programa es una función, lo cual es similar al tratamiento de cálculo λ anterior. Cada enunciado del lenguaje será una función, y elaboraremos el modelo de la ejecución de enunciados sucesivos determinando cuál debe ser la función de composición de los dos.

Se puede escribir una ecuación de tipos de datos para cada tipo de enunciado:

$Enun =$	$(Id \times Exp)$	Dominio de asignaciones
	$+(Enun \times Enun)$	Dominio de series
	$+(Exp \times Enun \times Enun)$	Dominio de condicionales
	$+(Exp \times Enun)$	Dominio de iteraciones

Como con cualquier intérprete, es necesario entender la computadora virtual subyacente. Se supondrá que cada identificador se refiere a una localidad específica en la memoria. La creación de seudónimos, el paso de parámetros y los apuntadores no se permiten en este ejemplo sencillo. Por consiguiente, para cada identificador hay una localidad única en la memoria que contiene su valor. En otras palabras, podemos elaborar un modelo de nuestro concepto de memoria

como una función que regresa, para cada identificador, su valor único. A una función así le llamamos *almacén*.

El almacén para un programa es una función que hace corresponder cada localidad (o *id*) con su valor apropiado. El efecto de ejecutar un enunciado es crear un almacén ligeramente modificado. Se tiene una nueva función (similar a la anterior) que ahora hace corresponder cada *id* con un valor, posiblemente el mismo o uno distinto. Por tanto, estamos viendo la ejecución como la composición de la función de almacén de valores seguida de la función que describe la ejecución de un enunciado particular.

Vemos un *almacén* como una correspondencia de identificadores a valores almacenables con una signatura de tipo $id \rightarrow valor$, donde *id* es el conjunto de identificadores en el lenguaje. A esto le llamaremos el *estado del programa*.

Al definir la semántica para un lenguaje de programación, existen tres funciones básicas que es necesario representar:

1. Es necesario describir la semántica de un *programa*. En este caso, la sintaxis de un programa define una función que tiene el efecto de hacer corresponder un número con otro número. En otras palabras, se busca una función M con signatura:

$$\mathcal{M} : prog \rightarrow [núm \rightarrow núm]$$

2. Es necesario describir un enunciado en el lenguaje. Cada enunciado hace corresponder un estado dado con otro estado, o:

$$\mathcal{C} : enun \rightarrow [estado \rightarrow estado]$$

Esto dice simplemente que cada enunciado sintáctico (dominio de $\mathcal{C}$) es una función de *estado* a *estado*. Por consiguiente, cada enunciado único toma un estado de programa (o correspondencia de identificador a valor) y produce una nueva correspondencia de identificador a valor que es el resultado de todos los enunciados previos, incluido este nuevo enunciado.

3. La función de enunciado $\mathcal{C}$ depende del valor de diversas expresiones, por consiguiente es necesario entender los efectos de la evaluación de expresiones en nuestro lenguaje. Cada expresión es una entidad sintáctica en la que intervienen identificadores, y cada uno tiene acceso al almacenamiento para producir un valor. Por tanto, a partir del conjunto de expresiones sintácticas, un conjunto de valores almacenables produce un valor de expresión. Esto proporciona la signatura para evaluación de expresiones como:

$$\mathcal{E} : exp \rightarrow [estado \rightarrow eval]$$

Como un ejemplo sencillo, si *exp* es la expresión $a+b$, entonces la función de evaluación es $\mathcal{E}(a + b) : estado \rightarrow eval$, y la aplicación de esta función al estado e proporciona la función:

$$\mathcal{E}(a + b)(e) = a(e) + b(e)$$

Por lo común se escribe $\mathcal{E}(a + b)$ como $\mathcal{E}\{\{a + b\}\}$ poniendo los argumentos sintácticos entre llaves contraídas en vez de paréntesis.

Ahora podemos elaborar el modelo de los dominios de nuestro lenguaje de programación.

estado = id → valor	*estados del programa*
id	*identificadores*
valor = eval	*valores*
eval = num + bool	*valores de expresiones*
núm	*enteros*
bool	*booleanos*
exp	*expresiones*
enun	*enunciados*

Para definir nuestro lenguaje, es necesario definir una función del tipo *estado → estado* para cada uno de los tipos de enunciado sintáctico. Para facilidad de lectura se usará una construcción *let* (sea) tipo l (similar a la let en ML). El término:

$$\text{let x} \leftarrow \text{a en } \textit{cuerpo}$$

se usará para expresar $((x : \textit{cuerpo})(a)$ y tiene un significado similar a la expresión λ:

$$(\lambda x.\ \textit{cuerpo}\ a)$$

Si x es del tipo de dominio $D \rightarrow D'$, entonces la expresión $x[v/e]$ se define como:

$$\text{x[v/e]} = \text{(D d) D': } \textbf{if } \text{d=v } \textbf{then } \text{e } \textbf{else } \text{x(d)}$$

Esto tiene el significado intuitivo de cambiar el componente v de x a e y representa el modelo básico de una "asignación".

Semántica de enunciados

Ahora proporcionamos la definición semántica para cada enunciado de nuestro lenguaje en términos de transformaciones sobre el *estado* de un cómputo.

begin enunciado end: La secuencia **begin...end** no tiene efecto alguno sobre el estado interno de un cómputo, de modo que es en efecto una función de identidad sobre el espacio de estado. Esto se representa como:

$$\mathcal{C}\{\{\textit{begin enun end}\}\} = \mathcal{C}\{\{\textit{enun}\}\}$$

composición: En este caso, sería deseable aplicar el estado resultante después de ejecutar $enun_1$ sobre $enun_2$. Esto se representa como composición de funciones:

$$\mathcal{C}\{\{enun_1 + enun_2\}\} = (\textit{estado } e)\textit{estado} : \mathcal{C}\{\{enun_2\}\}(\mathcal{C}\{\{enun_1\}\}(e))$$

El argumento para $\mathcal{C}\{\{enun_2\}\}$ es precisamente el estado que resulta después de evaluar $enun_1$.

asignación: Crear un nuevo mapa de almacenamiento resultante de evaluar *exp* en el estado actual:

$$\mathcal{C}\{\{id \leftarrow exp\}\} = (estado\ e)estado : ((valor\ v)estado : e[id/v])(\mathcal{E}\{\{exp\}\}(e))$$

if: Esto determina simplemente cuál de las dos funciones se deberá evaluar, evaluando primero $\mathcal{E}$ en la expresión, aplicando esto a la función booleana, la cual evalúa luego $enun_1$ o $enun_2$ según sea apropiado:

$$\begin{aligned}&\mathcal{C}\{\{\textit{if } exp \textit{ then } enun_1 \textit{ else } enun_2\}\} = \\ &\quad (estado\ e)estado : ((bool\ b)estado \rightarrow estado : \\ &\quad (\textbf{if } b \textbf{ then } \mathcal{C}\{\{enun_1\}\})\textbf{else } C\{\{enun_2\}\})(\mathcal{E}\{\{exp\}\}(e))(e)\end{aligned}$$

while: Esto requiere una definición recursiva (es decir, *rec*), puesto que la función **while** es parte de la definición:

$$\begin{aligned}&\mathcal{C}\{\{while\ exp\ do\ enun\}\} = \\ &\quad rec(estado\ e)estado : ((bool\ b)estado \rightarrow estado : \\ &\quad (\textbf{if } b \textbf{ then } \mathcal{C}\{\{enun\}\} \circ \mathcal{C}\{\{while\ exp\ do\ enun\}\}) \\ &\qquad \textbf{else } ((estado\ e')estado : e')) \\ &\quad (\mathcal{E}\{\{exp\}\}(e))(e)\end{aligned}$$

Adviértase que en la definición anterior para $\mathcal{C}\{\{while\}\}$, se usa de hecho $\mathcal{C}\{\{while\}\}$ en su definición (como parte de la expresión **then**). Esto explica el término *rec*, de *recursivo*, en la definición anterior. Así que el **while** es en realidad de la forma:

$$\mathcal{C}\{\{while\}\} = f(\mathcal{C}\{\{while\}\})$$

En términos más sencillos, se dice que el enunciado **while** es una solución a una ecuación de la forma:

$$x = f(x)$$

Estas soluciones se llaman *puntos fijos*, y lo que se desea es el punto fijo mínimo que resuelve esta ecuación. Un análisis completo de la teoría de puntos fijos está fuera del alcance de este libro; sin embargo, la descripción anterior proporciona una indicación de cómo se pueden ampliar los sistemas como el cálculo lambda para crear modelos de lenguajes de programación.

9.4.2 Verificación de programas

En la construcción de programas, cada vez nos preocupa más la corrección y la confiabilidad de nuestros productos terminados. Los lenguajes se están proyectando con características que mejoran estos atributos. Se puede aprovechar parte de la exposición anterior sobre semántica de lenguajes como ayuda en cuestiones de corrección. La corrección de un programa se puede mirar de tres maneras:

1. Dado el programa P, ¿qué significa? Es decir, ¿qué es su especificación E?
2. Dada la especificación E, desarrollar el programa P que implementa esa especificación.

3. ¿Llevan a cabo la misma función la especificación E y el programa P?

La primera de estas condiciones es simplemente la cuestión de modelos semánticos de la sección precedente. La segunda condición es tan sólo el problema de cómo construir un buen programa a partir de su especificación (es decir, el problema fundamental de la ingeniería de software en la actualidad), y la tercera condición es el problema medular en la verificación de programas (esto es, ¿significan lo mismo el programa y su especificación?). Aunque esta pregunta se puede plantear de tres maneras distintas, todas ellas son similares.

Desde 1965 hasta 1975, era popular estudiar la pregunta 3; la pregunta 1 fue popular desde alrededor de 1975 hasta finales de los años ochenta, y la pregunta 2 es actualmente de gran interés. Sin embargo, se necesitan técnicas similares para trabajar sobre las tres cuestiones.

Poner a prueba un programa no puede asegurar que esté libre de errores, excepto en casos muy simples. Un programa se pone a prueba con algunos conjuntos de datos de prueba y los resultados se comparan con los datos de entrada. Si los datos son consistentemente correctos para un número considerable de casos de prueba, entonces por lo común se dictamina que el programa está "depurado". Sin embargo, la verdad es que sólo de aquellos conjuntos de datos de prueba efectivamente ensayados se sabe que trabajan de manera correcta. El programa todavía puede tener errores para otros conjuntos de datos de entrada. Poner a prueba todos los conjuntos posibles de datos de entrada es ordinariamente imposible. Por tanto, el evaluador debe quedar satisfecho con una garantía incompleta de que el programa es correcto. Si se pudiera encontrar algún método que garantizara la corrección de un programa sin depender de las pruebas, entonces los programas se podrían hacer consistentemente más fiables.

Un programa hace el cómputo de una función. El programador necesita saber qué función debe computar el programa. Supóngase que se hiciera una especificación independiente de la función, como parte del diseño inicial del programa. Y supóngase que, a partir del programa, fuera posible determinar cuál es la función que el programa calculó en efecto. Si se pudiera demostrar de hecho que el programa computa exactamente la función especificada, entonces se habría probado que es correcto sin recurrir a ensayos.

El *cálculo de predicados* es una notación derivada de la lógica formal que es particularmente útil para especificar funciones complejas con precisión. Varias estrategias para probar la corrección de los programas se apoyan en la idea básica de que la función deseada se especifica usando el cálculo de predicados, y el programa se analiza luego para determinar si la función que computa es de hecho la que especifica la fórmula de cálculo de predicados.

La notación que se usa en general es $\{P\}S\{Q\}$, la cual significa que si el predicado P es cierto antes de la ejecución del enunciado S, y si S se ejecuta y termina, entonces Q será cierto después de la terminación de S. Si se agregan ciertas reglas de inferencia, se puede incorporar esta estructura en pruebas de cálculo de predicados, como lo demuestra la tabla siguiente:

```
                {B ≥ 0 }
1.      MULT(A,B) ≡{
2.          a := A;
3.          b := B;
4.          y := 0;
5.          while b>0 do
                begin
6.              y := y+a;
7.              b := b-1;
                end}
        {y = A × B}
```

Figura 9.9. Programa para el cómputo de $y = A \times B$.

Regla	**Antecedente**	**Consecuente**
1. *consecuencia*$_1$	$\{P\}S\{Q\}.(Q \Rightarrow R)$	$\{P\}S\{R\}$
2. *consecuencia*$_2$	$(R \Rightarrow P).\{P\}S\{Q\}$	$\{R\}S\{Q\}$
3. *composición*	$\{P\}S_1\{Q\}.\{Q\}S_2\{R\}$	$\{P\}S_1; S_2\{R\}$
4. *asignación*	$x := expr$	$\{P(expr)\}x := expr\{P(x)\}$
5. *si*	$\{P \wedge B\}S_1\{Q\}.$ $\{P \wedge \neg B\}S_2\ \{Q\}$	$\{P\}$**if** B **then** S_1 **else** $S_2\{Q\}$
6. *While*	$\{P \wedge B\}S\{P\}$	$\{P\}$**while** B **do** $S\{P \wedge \neg B\}$

Si se puede demostrar que el antecedente de cada regla de inferencia es cierto, entonces se puede reemplazar el antecedente con el consecuente. Esto permite incorporar pruebas de construcciones de programa dentro de nuestro conocimiento del cálculo de predicados.

La verificación axiomática funciona típicamente retrocediendo a lo largo de un programa. Si se sabe cuál es la poscondición para un enunciado, dedúzcase cuál precondición debe ser cierta para que el enunciado se ejecute. Por ejemplo, si la poscondición para el enunciado de asignación $X := Y + Z$ es $X > 0$, entonces, por el axioma de asignación:

$$\{P(expr)\}x := expr\{P(x)\}$$

donde $P(x) = X > 0$, se tiene que:

$$\{Y + Z > 0\}X := Y + Z\{X > 0\}$$

De modo que la precondición para el enunciado de asignación debe ser $Y + Z > 0$. Si se continúa en esta forma se pueden probar (con cierta dificultad) series de enunciados que contienen enunciados de asignación, **if** y **while**.

La regla de prueba para el enunciado **while** es quizá la más interesante:

if $\{P \wedge B\}S\{P\}$ then

$\{P\}$**while** B **do** $S\{\ P \wedge \neg B\}$

El predicado P se conoce como una *invariante* y debe ser cierto antes y después del cuerpo de la iteración **while**. Aunque hay heurísticas para encontrar invariantes, el problema general es imposible de resolver (véase la sección 9.3.3). Sin embargo, para muchos programas razonables, se pueden determinar invariantes.

El ejemplo 9.1 presenta la prueba del programa *MULT* de la figura 9.9, que hace el cómputo de $y = A \times B$. En este caso, si se estudia el programa se ve que cada paso a través de la iteración **while** suma a a y mientras el producto $a \times b$ disminuye en a, puesto que b se reduce en 1. Por tanto, se obtiene una invariante $y + a \times b = A \times B \wedge b \geq 0$.

La condición $\{P\}S\{Q\}$ expresa que *si* S termina, y si P es cierto antes de la ejecución de S, entonces Q será cierto después de que S termine. Para la asignación y el enunciado **if** no hay problemas. Se supone que ambos terminan. Pero esto no se cumple necesariamente para los enunciados **while**. Para probar que las iteraciones terminan, se necesita una prueba externa de este hecho. El método que se usa en general es como sigue:

1. Demuéstrese que existe alguna función entera f tal que $f > 0$ siempre que la iteración se ejecuta.

2. Si f_i representa la función f durante la i^a ejecución de la iteración, entonces $f_{i+1} > f_i$.

Si se puede demostrar que ambas condiciones son ciertas, entonces la iteración debe terminar. (Véase el problema 11 al final del capítulo.)

Esta notación sólo puede ser práctica si se puede automatizar el proceso de probar la corrección del programa; la aplicación de los métodos de prueba desarrollados en estos estudios tiende a ser tediosa para programas reales. Un error en la prueba puede hacer que parezca que el programa es correcto cuando de hecho no lo es. Por estas razones, es deseable automatizar el proceso de prueba de modo que, a partir de la especificación funcional del programa y del programa mismo, un sistema automático de prueba de programas pueda deducir la prueba completa con poca o ninguna intervención por parte de seres humanos.

En la actualidad se entiende de manera generalizada que los métodos de prueba de programas son útiles principalmente durante el diseño de un programa, de modo que, en cierto sentido, el programa se prueba conforme se escribe. Se ha encontrado que el uso de pruebas de programa para programas que han sido escritos en formas ordinarias, sin estar estructurados de manera que sea fácil probar que son correctos, es difícil o imposible. Los métodos para prueba de programas, aunque son de utilidad, no son lo bastante poderosos para tratar con estructuras de programa extremadamente complejas como las que se encuentran en programas más antiguos que han sido modificados una y otra vez durante su uso en producción. Los métodos de prueba de programas se enseñan por lo común a programadores en ejercicio de su profesión, y también forman parte de muchos cursos de introducción a la programación.

El impacto de este trabajo ha tenido su efecto sobre el diseño de lenguajes. Las características de lenguaje que inhiben la prueba de la corrección de los programas se consideran ahora en general como indeseables si se dispone de un sustituto adecuado que se avenga mejor a la prueba de corrección. Por ejemplo, el trabajo sobre pruebas de corrección ha ayudado a demostrar lo indeseable de la *creación de seudónimos* en el diseño de lenguajes, como se expuso en la sección 7.2.1. Ciertos lenguajes, como C++, incluyen una capacidad **assert**

EJEMPLO 9.1. Prueba axiomática del programa MULT.

Por lo general funciona en retroceso, deduciendo el antecedente del consecuente.
Necesario desarrollar "invariantes" para la iteración **while** en renglones 5-7:
y aumenta en a conforme b disminuye en 1. Esto da una invariante de:

	Enunciado	Razón
a	$\{y + a(b-1) = AB \wedge (b-1) \geq 0\}$ $b := b - 1\{y + ab = AB \wedge b \geq 0\}$	*asignación* (renglón 7)
b	$\{y + ab = AB \wedge b - 1 \geq 0\}y := y + a$ $\{y + a(b-1) = AB \wedge b - 1 \geq 0\}$	*asignación* (renglón 6)
c	$\{y + ab = AB \wedge b - 1 \geq 0\}$ $y := y + a; b := b - 1$ $\{y + ab = AB \wedge b \geq 0\}$	*composición* (a, b)
d	$(y + ab = AB) \wedge (b \geq 0) \wedge (b > 0) \Rightarrow$ $(y + ab = AB) \wedge b - 1 \geq 0$	*teorema*
e	$\{y + ab = AB \wedge (b \geq 0) \wedge (b > 0)\}$ $y := y + a; b := b - 1$ $\{y + ab = AB \wedge b \geq 0\}$	*consecuencia*$_2$ (c,d)
f	$\{y + ab = AB \wedge b \geq 0\}$**while** $\cdots$ $\{y + ab = AB \wedge b \geq 0 \wedge \neg b > 0\}$	*while* (renglón 5, e)
g	$\{0 + ab = AB \wedge b \geq 0\}$ $y := 0\{y + ab = AB \wedge b \geq 0\}$	*asignación* (renglón 4, f)
h	$\{0 + aB = AB \wedge B \geq 0\}b := B$ $\{0 + ab = AB \wedge b \geq 0\}$	*asignación* (renglón 3, g)
i	$\{0 + AB = AB \wedge (B \geq 0)\}a := A$ $\{0 + aB = AB \wedge B \geq 0\}$	*asignación* (renglón 2, h)
j	$B \geq 0 \Rightarrow 0 + AB = AB \wedge B \geq 0$	*teorema*
k	$\{B \geq 0\}a := A\{0 + AB = AB \wedge B \geq 0\}$	*consecuencia*$_2$
l	$\{B \geq 0\}a := A; b := B; y := 0$ $\{y + ab = AB \wedge b \geq 0\}$	*composición* (k, h, g)
m	$\{B \geq 0\}MULT(A, B)$ $\{y + ab = AB \wedge b \geq 0 \wedge \neg b > 0\}$	*composición* (l, f)
n	$(y + ab = AB) \wedge (b \geq 0) \wedge \neg(b > 0) \Rightarrow$ $(b = 0) \wedge (y = AB)$	*teorema*
o	$\{B \geq 0\}MULT(A, B)\{y = AB\}$	*consecuencia*$_1$ (m,n)

También debe demostrar que el programa termina. Esto se reduce a demostrar que todos los enunciados **while** terminan. La manera usual es:

1. Demostrar que hay cierta propiedad que siempre es positiva en la iteración (p. ej., $b \geq 0$).
2. Demostrar que esta propiedad disminuye en la iteración (p. ej., $b := b - 1$).

Para que ambas propiedades sigan siendo ciertas, la iteración debe terminar.

(aserto) que se basa en alguna medida en este método axiomático (véase la sección 9.1.1). El aserto se pone en el programa fuente y el programa *pone a prueba* el aserto conforme se ejecuta. No es una prueba de corrección *a priori*, pero la inclusión del aserto permite localizar muchos errores durante la vida posterior del programa.

9.4.3 Tipos de datos algebraicos

Las operaciones de reescritura y unificación de términos, ya descritas, desempeñan un importante papel en el desarrollo del modelo del tipo algebraico de datos. Si se describe la relación entre una serie de funciones, se afirma que cualquier implementación que se apegue a esa relación es una implementación correcta.

Por ejemplo, una *pila* de enteros se puede definir por medio de las operaciones siguientes:

$$\begin{array}{ll} \textit{agregar}: & \textit{pila} \times \textit{entero} \rightarrow \textit{pila} \\ \textit{remover}: & \textit{pila} \rightarrow \textit{pila} \\ \textit{tope}: & \textit{pila} \rightarrow \textit{entero} \cup \{\textit{sin definir}\} \\ \textit{vacía}: & \textit{pila} \rightarrow \textit{booleano} \\ \textit{tamaño}: & \textit{pila} \rightarrow \textit{entero} \\ \textit{pilanueva}: & \rightarrow \textit{pila} \end{array}$$

agregar y *remover* tienen la interpretación usual, *tope* regresa el valor en el tope de la pila sin eliminar la entrada, *vacía* verifica en busca de una pila vacía, *tamaño* es el número de elementos en la pila y *pilanueva* crea un nuevo ejemplar de pila.

Generación de axiomas algebraicos

Dado el conjunto de operaciones que especifican un tipo de datos, se pueden proporcionar varias heurísticas para desarrollar relaciones entre estas operaciones. Podemos dividir estas operaciones en tres clases: generadoras, constructoras y funciones:

Generadoras. Para el tipo abstracto de datos x, una generadora g construye un nuevo ejemplar del tipo. Por tanto, tiene la signatura:

$$g : \textit{not\_x} \rightarrow x$$

(*not_x* es cualquier tipo excepto x.) En nuestro ejemplo *pila*, *pilanueva* es una generadora.

Constructoras. Las constructoras c modifican ejemplares de nuestro tipo abstracto x y tienen signaturas:

$$c : x \times \textit{not\_x} \rightarrow x$$

agregar es una constructora en este ejemplo.

Funciones. Todas las demás operaciones sobre el tipo abstracto x son funciones. En nuestro ejemplo *pila*, *tope*, *remover*, *tamaño* y *vacía* son funciones.

Intuitivamente, se dice que cualquier objeto y del tipo abstracto x se puede crear por una aplicación de una generadora y aplicación repetida de constructoras. Para pilas, esto afirma

que cualquier pila es el resultado del uso de *pilanueva* para crear una pila vacía seguido de aplicaciones repetidas de *agregar* para poner cosas en la pila. Por ejemplo, la pila que contiene 1,5,3 se puede crear como:

$$agregar(agregar(agregar(pilanueva,1),5),3)$$

Si bien no existe un modelo formal para desarrollar axiomas algebraicos, el modelo heurístico siguiente funciona bien. Genérese un axioma para cada función con cada constructora y generadora. Puesto que nuestro ejemplo *pila* tiene tres funciones, una constructora y una generadora, esto produce seis axiomas. Una vez escrito el lado izquierdo del axioma, por lo común es fácil determinar cuál es su valor. Por ejemplo, con la función *tope* y la constructora *agregar*, el axioma se ve así:

$$tope(agregar(S,I)) = \ldots$$

Intuitivamente, se desea que el tope de la pila sea lo que se acaba de agregar a la pila, de modo que se obtiene el axioma:

$$tope(agregar(S,I)) = I$$

Si se continúa en esta forma, se obtienen los axiomas siguientes:

1 : $remover(pilanueva) =$ *pilanueva*
2 : $remover(agregar(S,I)) =$ S
3 : $tope(pilanueva) =$ *sin definir*
4 : $tope(agregar(S,I)) =$ I
5 : $vacía(pilanueva) =$ *cierto*
6 : $vacía(agregar(S,I)) =$ *falso*
7 : $tamaño(pilanueva) =$ 0
8 : $tamaño(agregar(S,I) =$ $tamaño(S) + 1$

Suele pensarse en estos axiomas como en reglas de reescritura. Es posible tomar un ejemplar de un objeto y usar las reglas para reescribir el ejemplar de una manera más sencilla. De acuerdo con la especificación, la operación *agregar* regresa un ejemplar de *pila* construido a partir de otro ejemplar de *pila* y algún *entero*. Por ejemplo, nuestra consulta se podría formar llamando *pilanueva*, aplicando luego un *agregar*, luego otro *agregar*, luego una *remover* y por último la función *vacía*, en esta forma:

$$vacía(remover(agregar(agregar(pilanueva, 42),17)))$$

La unificación de esta expresión con el axioma 2 proporciona una simplificación de la composición más exterior de *remover* con *agregar*, y la expresión como:

$$vacía(agregar(pilanueva, 42))$$

En este punto, se puede usar el axioma 6 y simplificar la cadena a *falso*. Se pudo modificar la expresión a sólo generadoras y constructoras antes de aplicar la definición de *vacía* para obtener la respuesta *falso*. Esto ilustra el punto expresado en el análisis anterior referente a constructoras. Una expresión algebraica que emplea sólo constructoras y generadoras tendrá una forma *canónica* o *normal*.

EJEMPLO 9.2. Inducción de tipos de datos.

1. Se prueba que P(S) ≡ tamaño(agregar(S,X))>tamaño(S) usando los axiomas como reglas de reescritura hasta que no sean posibles más simplificaciones:

tamaño(agregar(S,X))>tamaño(S) - Hipótesis del teorema
tamaño(S)+1>tamaño(S) - Por el axioma 8

2. En este punto se requiere inducción de datos para demostrar que tamaño(S)+1>tamaño(S) para las generadoras pilanueva y agregar, es decir, se necesita demostrar que P(pilanueva) y P(agregar(S,i):

(a) Argumento base: Primero reemplace S por pilanueva y demuestre que el argumento base es válido:

0+1 > 0+0 - Hecho matemático simple
tamaño(pilanueva)+1>tamaño(pilanueva) - Por el axioma 7

lo cual demuestra que el argumento básico es cierto para S=pilanueva

(b) Argumento de inducción: Suponga que el teorema se cumple para S, es decir, tamaño(S)+1>tamaño(S). Demuestre que el teorema se cumple para S'=agregar(S,X):

tamaño(S)+1>tamaño(S) - Hipótesis inductiva
tamaño(S)+1+1>tamaño(S)+1 - Suma de 1 a ambos lados
tamaño(agregar(S,X))+1>tamaño(agregar(S,X)) -Por axioma 8

lo cual demuestra que el paso inductivo también es válido.

Conclusión: Se ha demostrado que:

tamaño(agregar(S,X))>tamaño(S)

es un resultado cierto que se cumple para todas las pilas S.

Inducción de tipos de datos

Dado el conjunto de axiomas (es decir, relaciones) entre el conjunto de operaciones de un tipo algebraico de datos, suele ser necesario verificar que determinadas propiedades son ciertas. Éstas se vuelven entonces requisitos en cualquier programa (por ejemplo, en C o Pascal) que implemente esta especificación. La *inducción de tipos de datos* es una técnica para verificar estas propiedades. Está relacionada estrechamente con la inducción natural de los enteros positivos.

Sea $P(y)$ cierto predicado que concierne a $y \in x$. ¿En qué condiciones será cierto P para todos los miembros de tipo x? En forma similar a la inducción natural, se puede demostrar que P es cierto para objetos primitivos del tipo y luego demostrar que, cuando estos objetos primitivos se usan para construir nuevos objetos del tipo, la propiedad P sigue siendo cierta.

Se define la *inducción de tipos de datos* como sigue:

1. Dado el tipo x con funciones generadoras f_i, constructoras g_i y otras funciones h_i, y predicado $P(y)$ para $y \in x$:
2. Demuéstrese que $P(f_i)$ es válido. Este es el caso base que demuestra que el valor de la generadora es correcto.
3. Supóngase que $P(y)$ es válido. Demuéstrese que esto implica que $P(g_i(y))$ es válido.

Esto permite ampliar el predicado P a todos los objetos creados por aplicación de una generadora seguida de aplicaciones de constructoras.

4. Se concluye, por tanto, que $P(y)$ para cualquier y de tipo x.

Para nuestro ejemplo de pila, es necesario demostrar que $P(pilanueva)$ y $P(agregar(x,i))$. Usando inducción de tipos de datos se puede demostrar que agregar un valor en una pila incrementa el tamaño de la misma, es decir:

$$tamaño(agregar(S,X)) > tamaño(S)$$

El ejemplo 9.2 proporciona la prueba de esto.

9.4.4 Resolución

El cálculo de predicados también ha desempeñado un papel importante en el desarrollo de los lenguajes, sobre todo de Prolog. Durante los años sesenta, FORTRAN era el lenguaje dominante para cálculos científicos, en tanto que LISP era el preferido para programas basados en inferencia y deducción. Puesto que LISP empleaba estructuras de lista como tipo de datos básico, los árboles eran un tipo de datos compuesto natural que manejaban los programas en LISP. Si se representa la regla $A\ B\ C \Rightarrow D$ en LISP, la estructura de datos natural es un árbol con D como raíz y A, B y C como hojas. Para probar que D es cierto se requiere probar primero los subárboles A, B y C. Como se explicó en la sección 6.3.3 al tratar el retroceso, LISP es un lenguaje natural para la construcción de esta clase de algoritmos.

En 1965, Robinson desarrolló el principio de resolución, cuya implementación en 1972 se convirtió en Prolog. Aunque no es necesario entender cabalmente la resolución para programar en Prolog, sí ayuda para comprender la teoría en la que se sustenta la ejecución de Prolog.

Si se tiene un conjunto de predicados $\{P_1, P_2,...\}$, el problema general consiste en determinar qué teorema se puede generar a partir de estos predicados. O, en general, si P y Q son predicados, entonces, ¿es $P \Rightarrow Q$ (P implica Q) un teorema?

Iniciaremos este desarrollo con la observación de que todo predicado se puede escribir en *forma clausal*:

$$P_1 \wedge P_2 \wedge ... P_n \Rightarrow Q_1 \vee ... Q_m$$

donde cada P_i y Q_i son términos. Los siguientes son ejemplos de formas clausales:

Predicado	Forma clausal
$A \Rightarrow B$	$A \Rightarrow B$
$A \wedge B$	$A \wedge B \Rightarrow cierto$
$A \vee B$	$cierto \Rightarrow A \vee B$
$\neg A$	$A \Rightarrow falso$
$A \wedge \neg B \Rightarrow C$	$A \Rightarrow B \vee C$

Se puede demostrar esto reduciendo cada predicado a la *forma normal disyuntiva* (es decir, como una serie de términos $\vee$). Todos los términos negados se convierten en antecedentes y los términos no negados en consecuentes de la forma clausal. Por ejemplo:

Disjunctive normal form	Clausal form
$A \vee B \vee \neg C \vee D$	$C \Rightarrow A \vee B \vee D$
$\neg A \vee \neg B \vee \neg C \vee D$	$A \wedge B \wedge C \Rightarrow D$

Esto es una consecuencia directa de las transformaciones siguientes:

1. $P \Rightarrow Q$ es equivalente a $\neg P \vee Q$, y
2. (*Ley de DeMorgan*) $\neg P \vee \neg Q$ es equivalente a $\neg(P \wedge Q)$.

Al agrupar entre sí todos los términos negados en la forma normal disyuntiva para un predicado, por la ley de De Morgan son equivalentes a la negación del "y" de todos los predicados. Por la transformación 1 se convierten en el antecedente de la forma clausal.

Considérese cualquier forma clausal:

$$P_1 \wedge P_2 \wedge \ldots P_m \Rightarrow Q_1 \vee Q_2 \vee \ldots Q_n$$

La resolución de este predicado general se hace demasiado compleja. Pero considérense los casos donde $n = 1$ o $n = 0$. A estos predicados se les llama *cláusulas de Horn* y tienen este formato:

$$P_1 \wedge P_2 \wedge \ldots P_m \Rightarrow Q_1$$
$$P_1 \wedge P_2 \wedge \ldots P_m$$

Si se invierte el orden de antecedente y consecuente, y se reemplaza Þ por :–, se obtienen los enunciados equivalentes:

$$(1)\ Q_1 :- P_1 \wedge P_2 \wedge \ldots P_m.$$
$$(2)\ :- P_1 \wedge P_2 \wedge \ldots P_m.$$

los cuales son las familiares cláusulas de Prolog. La primera representa una regla y la segunda representa una consulta. Si se considera el caso $m = 0$, entonces se tiene un hecho de Prolog: Q_1:– *cierto*, o simplemente:

$$(3)\ Q_1.$$

El principio de resolución sobre cláusulas de Horn afirma ahora que:

Dada la consulta $Q_1 \ldots Q_n$ y la regla P_0:—$P_1 \ldots P_m$ entonces, si se unifican Q_1 y P_0, o, como se ha escrito en la sección 6.3.2, $Q_1\sigma = P_0\sigma$, se obtienen consultas equivalentes.

Si unificamos Q_1 con P_0, nuestra consulta original es entonces equivalente a:

1. $(P_1 \ldots P_m Q_2 \ldots Q_n)\sigma = R_1 \ldots R_{m+n-1}$, si P_0 era una regla, o
2. $(Q_1 \ldots Q_n)\sigma = R_1 \ldots R_{n-1}$, si P_0 era un hecho ($m = 0$),

donde $R_i = P_j\sigma$ o $R_i = Q_j\sigma$. Esperamos que, en último término, todas las R_i se unifiquen con hechos de nuestra base de datos, y den por resultado *cierto* con un historial de transformaciones que proporciona los resultados de nuestra consulta.

Este es el comportamiento esencial de ejecución de Prolog. El objetivo del espacio de búsqueda de Prolog es unificar $Q_1 ... Q_n$, y Prolog es libre de elegir cualquier regla P de la base de datos como hipótesis para intentar la resolución sobre ella. Si esto tiene éxito, σ describe las respuestas a nuestra consulta, y si fracasa, es necesario intentar reglas alternativas P' para buscar una sustitución válida.

Capítulo 10

Lenguajes simples de procedimientos

Al describir un lenguaje de programación "típico" para demostrar las características comunes de los lenguajes, el lenguaje de procedimientos es el que más se acerca a él en este libro. Estos lenguajes se componen de una serie de procedimientos (subprogramas, funciones o subrutinas) que se ejecutan cuando se les llama. Cada procedimiento consiste en una serie de enunciados, donde cada enunciado manipula datos que pueden ser locales al procedimiento, un parámetro que se pasa al interior desde el procedimiento de llamada o definidos de manera global.

Los datos locales para cada procedimiento se guardan en un registro de activación asociado con ese procedimiento, y los datos guardados en estos registros de activación tienen de manera habitual tipos de datos relativamente simples, como enteros, reales, de caracteres o booleanos.

En este capítulo se estudian dos de estos lenguajes, FORTRAN y C. COBOL todavía se usa mucho en aplicaciones de negocios, pero su empleo está decayendo y su interés dentro de la comunidad de la ciencia de la computación parece ser nulo. FORTRAN se caracteriza por asignación estática de almacenamiento (los registros de activación se pueden crear durante la traducción del lenguaje), aunque el más nuevo FORTRAN 90 cambia algo de esto, mientras que C se caracteriza por la creación dinámica de registros de activación. FORTRAN y C fueron proyectados para eficiencia en tiempo de ejecución. Aunque los tres lenguajes desempeñan papeles muy diferentes con respecto a sus comunidades de usuarios, todos tienen características similares en cuanto a computadora virtual y ejecución.

10.1 FORTRAN

El FORTRAN es un lenguaje que se usa ampliamente para el cómputo científico y de ingeniería. Ha evolucionado mucho en sus 40 años de vida, se le ha calificado como obsoleto e irrelevante numerosas veces, y sin embargo sigue entre nosotros y continúa en evolución. En esta sección enfocaremos nuestra atención al FORTRAN 77, que es la versión dominante en uso hoy día; sin embargo, indicaremos cómo el FORTRAN 90 ha ampliado y modificado el lenguaje.

La época actual representa un momento crítico en la evolución del FORTRAN. El FORTRAN 90 se aparta de manera radical de las versiones anteriores del lenguaje. Anteriormente, el FORTRAN se caracterizaba por asignación estática de memoria sin necesidad de asignación de registros de activación en tiempo de ejecución. Sin embargo, el FORTRAN 90 introduce

estructuras de control de enunciados múltiples, alcances anidados, tipos, recursión, arreglos dinámicos y apuntadores. En muchos aspectos, tiene las características de un lenguaje como Pascal con la sintaxis de FORTRAN. Todavía está por verse si el FORTRAN 90 representa un renacimiento del lenguaje con características modernas o una última boqueada de un lenguaje medieval que se desvanece lentamente en el "ocaso de los lenguajes de programación", al igual que los lenguajes previamente descartados como ALGOL, SNOBOL4 y muchos otros.

10.1.1 Historia

El FORTRAN fue el primer lenguaje de programación de alto nivel en alcanzar un uso amplio. Fue desarrollado inicialmente en 1957 por la IBM para ejecutarse en la computadora IBM 704. En esa época la utilidad de cualquier lenguaje de alto nivel estaba abierta al cuestionamiento por parte de programadores formados en programación de lenguaje ensamblador. Su queja más seria se refería a la eficiencia de ejecución del código compilado a partir de programas en lenguajes de alto nivel. Como consecuencia, el diseño de las primeras versiones de FORTRAN estaba fuertemente orientado a proporcionar eficiencia de ejecución. El éxito de este primer FORTRAN y su dependencia de características orientadas hacia una ejecución eficiente en la computadora IBM 704 han constituido un problema para el lenguaje, como más adelante señalamos. La primera definición estándar del lenguaje fue adoptada en 1966, y se hizo una revisión importante de este estándar en los años setenta, la cual condujo al FORTRAN 77. El FORTRAN 77 continuó con la tradición del patrón original al permitir que todos los datos se asignaran de manera estática.

La revisión actual del estándar, el FORTRAN 90, cambia radicalmente el foco de FORTRAN al permitir datos dinámicos. Además, agrega los conceptos de *obsolescencia* y *desaprobado* como indicaciones de características que se pueden omitir en una futura revisión de este estándar, una desviación radical respecto a la mayoría de los estándares, los cuales requieren compatibilidad con versiones anteriores del estándar.

10.1.2 Hola Mundo

FORTRAN se implementa en general usando tecnología tradicional de compiladores. Se usa un editor de texto para crear el programa, un compilador de FORTRAN traduce el programa a forma ejecutable, se usa un vinculador para fusionar subprogramas, el programa principal y rutinas de biblioteca en tiempo de ejecución en un programa ejecutable, y un paso de ejecución lleva a cabo el programa traducido.

En esta sección, y en las posteriores que describen otros lenguajes, usamos el mandato `edit` para referirnos al editor de texto del sistema, `fortran` (u otro nombre de lenguaje apropiado) como mandato para invocar el traductor, y `execute` para representar la ejecución del programa. Nos referiremos a todas las entradas de usuario usando tipo `fijo` y la respuesta de la computadora se muestra en letra *cursiva*. El carácter de señal de entrada del sistema es un %. Todo esto se necesita ajustar a la medida del propio ambiente operativo del lector.

Un programa en FORTRAN se compone de un programa principal y una serie de subrutinas. Para compilar y ejecutar un programa relativamente trivial y producir cierta salida, la figura 10.1 es representativa.

```
%edit trivial.for
      PROGRAM TRIVIAL
      INTEGER I
      I=2
      IF(I .GE. 2) CALL PRINTIT
      STOP
      END
      SUBROUTINE PRINTIT
      PRINT *, "Hola Mundo" '
      RETURN
      END
%fortran trivial.for
%execute
Hola Mundo
```

Figura 10.1. Impresión de "Hola Mundo" en FORTRAN.

10.1.3 Breve perspectiva del lenguaje

El diseño del FORTRAN se centra en el objetivo de eficiencia de ejecución. Las estructuras del lenguaje son simples en general y gran parte del diseño es más bien poco elegante, pero se consigue la meta de eficiencia de ejecución. Al analizar el FORTRAN podemos casi considerar que FORTRAN 77 y FORTRAN 90 son lenguajes distintos. El FORTRAN 90 incorpora casi todas las características modernas de datos y control, de las que carece el FORTRAN clásico, para conferirle el poder de lenguajes como C y Pascal.

Un programa en FORTRAN se compone de un programa principal y un conjunto de subprogramas, cada uno de los cuales se compila por separado de todos los demás, vinculando los programas traducidos en la forma ejecutable final durante el proceso de carga. Cada subprograma se compila en un segmento de código y registro de activación asignados en forma estadística. No se suministra gestión de almacenamiento en tiempo de ejecución; todo el almacenamiento se asigna de manera estática antes de que se inicie la ejecución del programa, aunque FORTRAN 90 cambia el modelo de ejecución para dar cabida a almacenamiento dinámico.

En FORTRAN sólo se suministra un conjunto restringido de tipos de datos; cuatro tipos de datos numéricos (enteros, reales, complejos y reales de doble precisión), datos booleanos (llamados *logical*), arreglos, cadenas de caracteres y archivos. Se provee un conjunto extenso de operaciones aritméticas y funciones matemáticas, lo que refleja la orientación del lenguaje hacia el cómputo científico y de ingeniería. Se proporcionan operaciones relacionales y booleanas así como selección simple en arreglos usando subindización. Se manejan archivos tanto secuenciales como de acceso directo, y se dispone de un conjunto flexible de recursos para entrada-salida y especificación de formatos.

La debilidad más grande del FORTRAN 77 está en sus restringidas facilidades para estructuración de datos, las cuales se limitan esencialmente a arreglos y cadenas de caracteres de longitud declarada fija. No se proporcionan medios para definiciones de tipos o abstracción de datos. Los subprogramas (procedimientos y funciones) proveen el único mecanismo de abstracción.

Las estructuras de control de secuencia incluyen expresiones con las operaciones infijas y prefijas y llamadas de función usuales. El control de secuencia de enunciados se apoya fuertemente en etiquetas de enunciado y enunciados GOTO, aunque cada revisión del lenguaje ha ido agregando más estructuras de control anidadas. FORTRAN 66 tenía una fuerte influencia de la arquitectura de computadora subyacente sobre la cual se ejecutaba. FORTRAN 77 incorporó ciertas estructuras de control modernas (por ejemplo, la condicional IF ... THEN ... ELSE), y el FORTRAN 90 amplió este concepto al grado de que ahora es posible escribir programas en FORTRAN sin enunciados GOTO. FORTRAN 90 incluye el concepto de la característica **deprecated** (desaprobado), como el enunciado IF aritmético que se considera obsoleto y se eliminará en una versión posterior del estándar del lenguaje. Puesto que casi todas las características de FORTRAN 66 que están basadas en la máquina en sí están desaprobadas ahora, para cuando se haga la próxima revisión del estándar FORTRAN deberá ser un lenguaje moderno.

Sólo se proporcionan dos niveles de entornos o ambientes de referencia, global y local; sin embargo, el lenguaje FORTRAN 90 agrega el concepto de subrutinas anidadas. El ambiente global se puede dividir en ambientes comunes individuales (llamados *bloques COMMON*, ahora en la lista desaprobada) que se comparten entre conjuntos de subprogramas, pero sólo se pueden compartir objetos de datos en esta forma. Los parámetros se transmiten de manera uniforme por referencia (o valor-resultado).

El lenguaje está pensado para un ambiente operativo y de programación de procesamiento por lotes. No se incluyen características especiales para manejar la construcción de programas grandes más allá de cláusulas para la compilación independiente de subprogramas.

Ejemplo anotado

El ejemplo que se muestra en la figura 10.2 presenta la sumatoria de un vector. Como una reliquia de sus raíces de procesamiento por lotes, las etiquetas numéricas de enunciados se dan en las columnas 2 a 5, y una C en la columna 1 indica un comentario. Todos los enunciados requieren un renglón a menos que la columna 6 tenga un carácter no en blanco, lo que indica una continuación del enunciado anterior.

Los espacios en blanco no se toman en cuenta, de modo que se pueden insertar espacios libremente en el programa fuente para aumentar la legibilidad, como la indicación de anidamiento de bloques de enunciados en la figura 10.2. Desafortunadamente, casi todos los programadores en FORTRAN continúan apegándose a la "regla" de que los enunciados comienzan en la columna 7. Esto es sólo una guía, y los programas buenos incluyen abundante "espacio en blanco" para una mejor legibilidad. Los números de renglón en serie a la izquierda de la figura 10.2 no son parte del programa; sirven sólo para el análisis subsiguiente.

Renglón 1. El nombre del programa es `PRINCIPAL`.

```
1         PROGRAM PRINCIPAL
2           PARAMETER (TAMMÁX=99)
3           REAL A(TAMMAX)
4     10    READ (5,100,END=999) K
5    100    FORMAT(I5)
6             IF (K.LE.O.OR K.GT.TAMMÁX) STOP
7             READ *,(A(I),I=1,K)
8             PRINT *,(A(I),I=1,K)
9             PRINT * ,, 'SUMA=', SUM(A,K)
10            GO TO 10
11    99    PRINT * ,, "Todo listo"
12          STOP
13          END
14  SUBPROGRAMA DE SUMATORIA EN C
15        FUNCTION SUM(V,N)
16          REAL :: V(N) ! Declaración de estilo nuevo
17          SUM = 0.0
18          DO 20 I = 1,N
19            SUM = SUM + V(I)
20    20      CONTINUE
21          RETURN
22          END
```

Figura 10.2. Ejemplo en FORTRAN para sumar un arreglo.

Renglón 2. TAMMÁX es una constante definida por el programador. El valor de 99 es de hecho lo que el traductor usa, no el nombre TAMMÁX, al traducir este programa.

Renglón 3. Se declara un arreglo real de tamaño 99, con límites que van de 1 a 99. Se supone que los límites inferiores son 1.

Renglón 4. Esto lee e introduce el tamaño del arreglo en K, la cual, puesto que no está declarada, es una variable entera. El 5 después del READ se refiere a un archivo de entrada. Esto se refiere a la entrada usual (de teclado). Un 6 se refiere al archivo usual de salida (visualización). El número que sigue se refiere a un enunciado FORMAT que proporciona el formato de los datos provenientes del teclado y que se usará para leer e introducir el valor de K. El END=999 opcional se refiere a una excepción de final de archivo. Si no hay más datos por leer, el control se transfiere al enunciado en la etiqueta 999.

Renglón 5. Este enunciado FORMAT dice que los datos tendrán formato de enteros (formato I) y ocupan 5 lugares en el renglón de entrada. El formato F es real "fijo". Por ejemplo, el número real 5.123 tendría formato F5.3, lo que significa que ocupa cinco caracteres y tres están a la derecha del punto decimal.

Renglón 6. Éste verifica si K está entre 1 y 99. Si no es así, el programa se detiene.

Renglón 7. Este enunciado READ lee e introduce el valor de los elementos de arreglo desde A(1) hasta A(K). En vez de usar un enunciado FORMAT, como en READ(5,101), el asterisco indica un READ dirigido por lista, el cual analiza sintácticamente números reales en orden provenientes del flujo de entrada. A la variable I sólo se le da un valor dentro de este enunciado.

Renglón 8. Esto reproduce los datos de entrada sobre la corriente de salida en un formato dirigido por lista.

Renglón 9. Se llama la función SUM y se imprime su valor, después de imprimir la cadena SUMA=.

Renglón 10. El FORTRAN 77 no tiene construcciones **while**. Este enunciado transfiere el control de vuelta a la etiqueta de enunciado 10 para leer e introducir el dato de arreglo siguiente. Los renglones del 4 al 10 forman en efecto la construcción:

```
while no_final_de_archivo do
    Procesar arreglo siguiente
end
```

Renglones 11-12. Imprimen Todo Listo en un formato dirigido por lista y terminan la ejecución.

Renglón 13. Fin de la unidad del programa principal.

Renglón 14. Un comentario. Estos pueden aparecer en cualquier punto del programa.

Renglones 15-22. Subprograma de función SUM. Esta función se compila por separado del programa principal. No se usa información proveniente del programa principal para pasar información al compilador. El renglón erróneo:

```
FUNCTION SUM(V,N,M)
```

también se compilaría, pero puede fallar cuando el cargador intenta fusionar este subprograma con el programa principal.

Renglón 16. Aunque el arreglo se da como V(N), se refiere al parámetro asignado estadísticamente, el arreglo real A(99) del renglón 3 cuando se usa en el renglón 9. Este renglón muestra el estilo de las declaraciones para FORTRAN 90 agregando el símbolo ::, y FORTRAN 90 también permite comentarios incrustados ¡usando el designador!

Renglón 18. Esta iteración DO fija I en 1, y luego incrementa I hasta que I es igual a N. La etiqueta de enunciado indica el final de la iteración. Si el incremento no es 1, entonces se puede proporcionar un tercer parámetro, como en:

```
DO 20 J=2,20,2
```

que pone en secuencia J a través de todos los enteros pares desde 2 hasta 20.

Renglón 19. En una función en FORTRAN se devuelve un valor asignando un valor al nombre de la función.

Renglón 20. CONTINUE es un enunciado nulo que simplemente guarda el lugar para una etiqueta de enunciado. Aquí termina la iteración DO.

Renglones 21-22. Devuelven al programa de llamada e indican el final del subprograma.

10.1.4 Objetos de datos

El lenguaje FORTRAN suministra cuatro tipos de datos numéricos: entero, real, real de doble precisión y complejo. También se incluye un tipo booleano (llamado asimismo *logical*). Los arreglos, cadenas de caracteres y archivos son los únicos tipos de datos estructurados.

Objetos de datos primitivos

Variables y constantes. Los nombres de variables van de 1 a 6 (31 en FORTRAN 90) caracteres de longitud, comienzan con una letra y contienen letras y dígitos (y _ en FORTRAN 90). El FORTRAN no es sensible a mayúsculas/minúsculas; PRINT, print, PrInT y PRint se refieren todas al mismo nombre. Tradicionalmente, el FORTRAN se escribía con sólo letras mayúsculas, pero la práctica actual consiste en mezclar letras mayúsculas y minúsculas, como en casi todos los demás lenguajes.

Los primeros FORTRAN no toman en cuenta los caracteres en blanco, pero éstos son significativos en FORTRAN 90. Las expresiones siguientes representan todas el mismo enunciado en FORTRAN 77:

```
DO 20 I = 1,20
DO20I=I,20
D O 2 0 I = 1 , 2 0
```

pero lo siguiente es un error en FORTRAN 90:

```
DO I = 1,20
...
ENDDO ! Espacio faltante entre END y DO
```

Las variables no se tienen que declarar explícitamente. Se puede dar una declaración explícita como:

```
REAL A, B, SUM
DOUBLE PRECISION Q, R
LOGICAL :: T
```

La definición de *T* muestra la sintaxis de FORTRAN 90 para declaraciones. Si no aparece una declaración explícita, una *convención de nomenclatura* con base en el primer carácter del nombre de la variable determina el tipo. La convención de nomenclatura por omisión especifica que los nombres que comienzan con I-N son variables *enteras*; todas las demás son variables *reales*. Sin embargo, el programador puede cambiar la convención de nomenclatura empleada en cualquier programa iniciando la definición del subprograma con un enunciado IMPLICIT. Por ejemplo, el enunciado:

```
IMPLICIT INTEGER (A-Z)
```

al principio del subprograma hace que se suponga el tipo *integer* (entero) para todas las variables no declaradas en forma explícita. La declaración:

IMPLICIT NONE

en FORTRAN 90 desactiva todas las declaraciones implícitas de variables e indica como error cualquier variable que se use sin una declaración. Es práctica recomendable poner esto en cada programa.

Las constantes definidas por el programador se pueden incluir mediante el uso de un enunciado PARAMETER al principio de un subprograma, por ejemplo:

```
PARAMETER (KMÁX=100, PTOMED=50)
```

El tipo implícito determinado por la convención de nomenclatura se aplica también a las constantes definidas, o se puede expresar de manera explícita (en FORTRAN 90):

```
REAL, PARAMETER :: EPSILON = .0012
```

La verificación estática de tipos se usa en FORTRAN, pero es incompleta. Muchas características de lenguaje, incluso argumentos en llamadas de subprograma y el uso de bloques COMMON, no se pueden verificar estáticamente, en parte porque los subprogramas se compilan en forma independiente. Las construcciones que no se pueden verificar de manera estática se dejan ordinariamente sin verificar durante la ejecución en implementaciones de FORTRAN.

Tipos de datos numéricos. Los tipos de datos numéricos *entero*, *real* y (*real*) *de doble precisión* se representan por lo común directamente usando representaciones de números de *hardware*. El tipo *complejo* se representa como un par de números reales, guardados en un bloque de dos palabras.

Se suministra un conjunto extenso de operaciones primitivas para aritmética y para conversión entre los cuatro tipos numéricos. Las operaciones aritméticas básicas (+, –, *, /) y la exponenciación (**) se complementan con un conjunto grande de *funciones intrínsecas* predefinidas que incluyen operaciones trigonométricas y logarítmicas (*sin*, *cos*, *tan*, *log*), raíz cuadrada (*sqrt*), máximo (*max*) y mínimo (*min*), así como funciones explícitas de conversión de tipos para los diversos tipos numéricos. También se suministran las operaciones relacionales usuales sobre valores numéricos, las cuales se representan como:

Operador	Significado	Operador	Significado
.EQ.	igual	.NE.	no igual
.LT.	menor que	.GT.	mayor que
.LE	menor que o igual	.GE.	mayor que o igual

Tipos de datos lógicos. El tipo booleano se llama LOGICAL, con las constantes .TRUE. y .FALSE. Las operaciones .NOT., .AND. y .OR. representan las operaciones booleanas básicas; .EQV. y .NEQV. representan equivalencia booleana y su negación, respectivamente.

Las expresiones LOGICAL se pueden construir en la forma usual con datos numéricos [por ejemplo, *(A.LT.7).OR.(B.GE.15)*].

Variables apuntador. El FORTRAN 77 no incluye variables apuntador, puesto que todos los datos se asignan de manera estática. Por consiguiente, la construcción de estructuras de lista en FORTRAN suele significar el uso de arreglos grandes en los que el índice del arreglo sirve como apuntador al elemento de lista siguiente. Esto funciona eficazmente cuando todas las estructuras asignadas son del mismo tipo. Sin embargo, es difícil escribir programas que creen estructuras de datos arbitrarias usando apuntadores sin violar algunas de las reglas del FORTRAN 77. El FORTRAN 90 no agrega apuntadores como nuevo objeto de datos tipificado.

Los apuntadores se declaran como:

```
INTEGER, POINTER :: P
```

lo cual indica que *P* es un apuntador a un entero. El valor r de *P* se fija en el valor l de X por medio de P=>X

X debe ser un *objetivo* (target) legal de un apuntador, especificado por:

```
INTEGER, TARGET :: X
```

El uso de un apuntador en una expresión es lo mismo que usar el valor de su valor r; es decir, el apuntador se desrreferencia automáticamente:

```
P1=>X
P2=>Y
P1=P2 ! Es lo mismo que X = Y
```

El almacenamiento dinámico para un apuntador *P*1 se asigna a través de un enunciado *ALLOCATE*(*P*1), y el almacenamiento se libera a través de *DEALLOCATE*(*P*1).

Tipos de datos estructurados

Arreglos. Los ámbitos de subíndices para arreglos se deben declarar en forma explícita, ya sea en un enunciado DIMENSION (el cual permite determinar el tipo de los componentes por la convención de nomenclatura implícita) o declarando el arreglo y su tipo de componentes mediante una de las declaraciones antes mencionadas, por ejemplo:

```
REAL M(20), N(-5:5)
DIMENSION I(20,20)
```

Si se omite el límite inferior de subíndices, se supone que es 1. El arreglo *I* precedente se supone que es INTEGER a menos que un enunciado IMPLICIT redefina la convención de nomenclatura para *I*. Un arreglo puede tener hasta siete dimensiones.

Se usan subíndices para seleccionar componentes de arreglos, empleando la misma sintaxis que para llamadas de función, por ejemplo, $M(3)$ o $N(I+2)$.

A diferencia de casi todas las definiciones de lenguajes, los arreglos en FORTRAN se guardan en *orden principal por columnas*. Es decir, una matriz se guarda en forma secuencial por columnas en vez de por filas. No se guarda un descriptor con un arreglo; si es necesario, se

guarda por separado. Por ejemplo, en el acceso a arreglo bidimensional $A(7,9)$ (con referencia a la sección 4.3.5), puesto que se conocen todos los límites del arreglo en tiempo de compilación, la fórmula de acceso:

$$valorl(A(7,9)) = OV + 7 \times S + 9 \times E$$

puede ser computada por el traductor, puesto que OV, S y E son todos conocidos y constantes.

Los arreglos se asignan estáticamente, al igual que todos los datos en FORTRAN, a menos que se proporcione el atributo ALLOCATABLE en FORTRAN 90.

Cadenas de caracteres. Se pueden declarar variables de cadena de caracteres de longitud fija; por ejemplo:

```
CHARACTER S*10, T*25
CHARACTER(LEN=7) U
```

define que S y T contienen cadenas de caracteres de longitud 10 y 25, respectivamente, y U como una cadena de siete caracteres en FORTRAN 90. La declaración IMPLICIT también se puede usar para proveer una longitud por omisión y un tipo CHARACTER para variables no declaradas explícitamente. También se pueden definir arreglos de cadenas de caracteres y funciones que devuelven cadenas de caracteres como sus valores. La operación de concatenación (//)permite unir dos cadenas de caracteres. Las operaciones relacionales (.EQ., .GT., etc.) también están definidas para cadenas de caracteres usando ordenamiento lexicográfico.

Las posiciones de los caracteres en el valor de una variable de cadena de caracteres están numeradas desde 1 hasta el límite declarado. Se puede seleccionar una subcadena de una cadena de caracteres usando la sintaxis:

NombreDeVariableDeCar(*PrimeraPosCar* : *ÚltimaPosCar*)

Se puede omitir el indicador de posición ya sea del primer o último carácter, y se usa el valor por omisión de *PrimeraPosCar* = 1 y *ÚltimaPosCar* = *LímiteDeclarado*.

También se pueden invocar las conversiones de formato que suministran las operaciones de entrada-salida (véase más adelante) para convertir una cadena de caracteres guardada en una variable o arreglo a representaciones de almacenamiento binario interno en otra variable o arreglo de tipo numérico o lógico (o viceversa). Se usan los mismos enunciados READ, WRITE y FORMAT, pero las especificación UNIT es ahora un nombre de variable; por ejemplo:

```
READ (UNIT=A, FMT=200) M, N
```

especifica que los datos se deben "leer" desde la variable de cadena de caracteres A, precisamente como si A fuera un archivo externo, convertirse a representación interna de enteros y guardarse en M y N.

Representación de almacenamiento

La representación de almacenamiento secuencial es decisiva en la definición de las declaraciones COMMON y EQUIVALENCE (ambas características desaprobadas en FORTRAN 90). COMMON se analiza más adelante porque su propósito principal es establecer

datos compartidos entre subprogramas. La declaración EQUIVALENCE permite que más de una variable simple o subindizada se refiera a la misma localidad de almacenamiento, como en:

```
REAL X
INTEGER X
EQUIVALENCE (X,Y)
```

Esta declaración especifica que los nombres de variables simples *X* y *Y* se van a asociar con la misma localidad de almacenamiento. El valor guardado en esa posición se puede recuperar o modificar más tarde a través de cualquiera de los identificadores *X* y *Y*. Esta característica es claramente propensa a errores. La declaración EQUIVALENCE, aunque útil en un lenguaje con sólo gestión de almacenamiento estático, es un sustituto pobre de la auténtica gestión de almacenamiento dinámico. En esencia, la carga de la asignación de almacenamiento y nuevo uso se deja al programador, y no se suministran salvaguardas.

Inicialización. Se usa un enunciado DATA para inicializar datos estáticos. El enunciado de inicialización de datos se compone de una lista de variables simples y subindizadas con una lista de valores iniciales por asignar. Por ejemplo:

```
DATA X/1.0/, Y/3.1416/, K/20/
```

define los valores iniciales de las variables *X*, *Y* y *K* como 1.0, 3.1416 y 20, respectivamente. *X*, *Y* y *K* deben ser variables locales; es decir, deben tener almacenamiento asignado estáticamente susceptible de ser inicializado al comienzo de la ejecución del programa.

Se pueden asignar valores iniciales a variables y arreglos globales (definidos en bloques COMMON), pero estas asignaciones se deben hacer usando el enunciado de inicialización de datos dentro de un *subprograma de datos de bloque* especial que consiste sólo en la cabecera especial BLOCK DATA, seguida de declaraciones COMMON y enunciados de inicialización de datos para las variables y arreglos COMMON. El resultado de la traducción de un subprograma de datos de bloque de este tipo es un conjunto de tablas de cargador que el cargador utiliza al final de la traducción para inicializar las localidades de almacenamiento apropiadas antes que se inicie la ejecución.

Tipos definidos por el usuario

El FORTRAN 77 carece de mecanismos para crear tipos de usuario. todos los tipos están predefinidos en el lenguaje. El FORTRAN 90 introduce un enunciado TYPE. Por ejemplo, para generar cadenas de longitud variable, se podría definir un tipo *CADENAV:*

```
TYPE CADENAV
    INTEGER :: SIZE
    CHARACTER(LEN=20):: LEN
END TYPE CADENAV
```

Las declaraciones son similares a otras declaraciones en FORTRAN 90:

TYPE(CADENAV) :: MICADENA

y % se usa como selector: *MICADENA%SIZE* o *MICADENA%LEN*.

10.1.5 Control de secuencia

El FORTRAN tiene un conjunto limitado de estructuras de control. Algunas de éstas muestran el paso del tiempo, por provenir de sus raíces iniciales que buscaban proporcionar una ejecución eficiente.

Expresiones

Una expresión se puede usar para el cómputo de un solo número, valor lógico o cadena de caracteres. La asociatividad de las operaciones primitivas es de izquierda a derecha para todas las operaciones excepto ** (exponenciación), la cual asocia de derecha a izquierda. Se pueden usar paréntesis para controlar el orden de evaluación en la forma usual. Las operaciones primitivas tienen este orden de precedencia (de mayor a menor):

```
**
/
+ -
//
.EQ. .NE. .LT. .GT. .LE. .GE.
.NOT.
.AND.
.OR.
.EQV. .NEQV.
```

Las funciones pueden no tener efectos colaterales que afecten el resultado de la evaluación de expresiones. La definición del FORTRAN permite explícitamente que cualquier implementación de FORTRAN reacomode el orden de evaluación de una expresión para cualquier tipo de optimización, siempre y cuando, desde luego, no se viole la estructura de árbol de la expresión definida por las reglas de precedencia y los paréntesis. Desafortunadamente, no hay manera de hacer valer la prohibición sobre efectos colaterales que afectan la evaluación.

Enunciados

Asignación. El enunciado de asignación tiene la sintaxis siguiente:

variable = *expresión*

donde el valor r de *expresión* se asigna al valor l de *variable*, por ejemplo:

```
X = Y + Z
A(1,2) = U + V – (A(I, J)
```

La *variable* y la *expresión* pueden ser ambas numéricas, lógicas, o de caracteres. La asignación a una variable de tipo CHARACTER causa que la cadena asignada se ajuste a la longitud declarada de la variable receptora, ya sea por truncamiento (si es demasiado larga) o extensión con espacios en blanco (si es demasiado corta).

Enunciado condicional. Se suministran cuatro enunciados condicionales de bifurcación, las construcciones IF tradicionales de FORTRAN 77 y también el enunciado CASE en FORTRAN 90.

Un enunciado IF *aritmético* proporciona una ramificación de tres vías, según sea el valor de una expresión aritmética, negativo, cero o positivo:

$$\text{IF}(expr)\ etiqueta_{valor\ neg},\ etiqueta_{valor\ cero},\ etiqueta_{valor\ pos}$$

donde las *etiqueta* son etiquetas de enunciado a las cuales se va a transferir el control. Esto se considera obsoleto y se puede eliminar de una versión futura del lenguaje. Los programas que tienen construcciones IF aritméticas son difíciles de seguir, puesto que las etiquetas de enunciado correspondientes pueden estar dispersas a través del programa, lo cual oculta la estructura lógica del mismo.

Un enunciado IF *lógico* sólo permite la ejecución de un único enunciado si el valor de la expresión lógica es cierto:

IF(*expr*) *enunciado*

El *enunciado* puede no ser un enunciado DO u otro IF lógico, lo que impide cualquier clase de estructura de enunciados anidados. El renglón 6 de la figura 10.2 muestra un uso representativo del IF lógico.

Un enunciado IF de *bloque* en FORTRAN 77 permite una alternancia *if ... then ... else ... endif* de enunciados sin el uso de etiquetas de enunciado. La forma es:

```
IF expresión_de_prueba THEN
      -sucesión de enunciados en renglones individuales
ELSE IF expresión_de_prueba THEN
      -sucesión de enunciados en renglones individuales
      ...
ELSE
      -sucesión de enunciados en renglones individuales
END IF
```

donde las partes ELSE IF o ELSE se pueden omitir. Adviértase que, a causa de la sintaxis de enunciado por renglón de FORTRAN, cada uno de los delimitadores IF, ELSE IF, ELSE y END IF debe aparecer como un renglón individual en el programa.

El enunciado CASE agregado en el FORTRAN 90 tiene esta sintaxis:

```
SELECT CASE expresión
CASE (selector_de_caso)
  bloque_de_enunciados
...– Otros casos
CASE DEFAULT
  bloque_de_enunciados
END SELECT
```

donde la evaluación de *expresión* debe dar un entero, una cadena de caracteres o un valor lógico y el *selector_de_caso* debe ser una constante o un ámbito de valores de constantes, como **case** (1 : 3,5 : 6,9), lo cual indica selección si la expresión tiene un valor de 1, 2, 3, 5, 6 o 9.

El caso DEFAULT es opcional y si ningún *selector_de_caso* concuerda con la *expresión*, el enunciado se pasa por alto.

Enunciado de iteración. El enunciado de iteración de FORTRAN 90 tiene la forma:

```
DO control_de_la iteración
   bloque_de_enunciados
   END DO
```

Si se omite *control_de_la iteración*, esto representa una iteración infinita. La salida de la iteración deberá ser a través de un enunciado EXIT.

Para una iteración con un número fijo de iteraciones, la sintaxis es:

DO *etiqueta var_ini* = *val_ini, val_final, incremento*

donde *var_ini* es una variable entera simple que se va a usar como contador durante las ejecuciones repetidas del cuerpo de la iteración, *val_ini* es una expresión que especifica el valor inicial del contador, *val_final* especifica de igual manera el valor final, el cual, cuando es alcanzado por el contador, termina la iteración, e *incremento* especifica la cantidad que se debe sumar al valor de *var_ini* cada vez que se recorre la iteración. Si se omite, el incremento es 1.

El cuerpo de la iteración abarca desde el enunciado DO hasta el enunciado marcado con *etiqueta*, en las primeras versiones de FORTRAN, o hasta un enunciado END DO en FORTRAN 90 (y se omite la *etiqueta*). Ni el valor de *var_ini* ni los valores de variables empleadas para especificar *val_ini*, *val_final* o *incremento* se pueden modificar durante la ejecución de la iteración. Esta restricción permite que todos los parámetros de la iteración se calculen una vez en la primera entrada a la iteración y nunca se repita el cómputo.

Las versiones antiguas de FORTRAN probaban la condición de terminación en una iteración DO sólo *después* de la primera ejecución del cuerpo de la iteración, con el resultado de que un cuerpo de la iteración DO siempre se ejecutaba al menos una vez, incluso si el valor inicial de la variable de iteración era menor que el valor final especificado en la primera entrada a la iteración. Sin embargo, el FORTRAN 77 prevé que la prueba se haga *antes* de que se ejecute el cuerpo por primera vez.

El enunciado EXIT de FORTRAN 90 provoca que el control salga del enunciado DO actual. El enunciado CYCLE de FORTRAN 90 hace que el control pase al extremo del enunciado DO actual e inicie la iteración siguiente.

Enunciado nulo. El enunciado CONTINUE es un enunciado nulo que se usa para reservar el lugar a una etiqueta de enunciado. Su uso principal es como el término de una iteración DO o como el objeto de un GOTO. Esto permite que la etiqueta de enunciado permanezca en un punto fijo, mientras que cualesquier otros enunciados se pueden modificar si es necesario.

Control de subprogramas. El enunciado CALL se usa para invocar un subprograma, y se da como:

CALL *nombre_de_subprograma*(*lista_de_parámetros*)

donde la lista de parámetros es una sucesión de nombres de parámetros reales o expresiones.

Los subprogramas de función son subprogramas que devuelven un valor y se usan simplemente en una expresión, como en 2 + *mifun*(1,23,.*TRUE.*). Las funciones devuelven valores asignándolos al nombre de la función antes de ejecutar el enunciado RETURN:

```
MIFUN = 27
RETURN
```

El nombre de la función se puede usar como variable local dentro del subprograma. El FORTRAN 77, con su estructura de almacenamiento estático, no permite recursión; el FORTRAN 90 da cabida a recursión y almacenamiento dinámico.

La ejecución del programa termina con un enunciado STOP.

Los subprogramas devuelven al programa de llamada con un enunciado RETURN.

Construcciones propensas a error

La dependencia del FORTRAN respecto a etiquetas de enunciado se basaba en el diseño antiguo del lenguaje, donde el lenguaje seguía la arquitectura de la computadora subyacente. Se sabía menos acerca del buen diseño de los programas, la necesidad de estructuras de control anidadas, el uso de tipos definidos por el usuario y la necesidad de encapsulamiento. Todas las construcciones siguientes se deben evitar al construir programas bien estructurados:

Enunciado GOTO. El enunciado **goto**:

GOTO *etiqueta*

transfiere el control al enunciado que tiene la etiqueta *etiqueta*. Este enunciado no puede hacer que el control entre en un DO o en un enunciado IF de bloque. Desafortunadamente, puesto que los FORTRAN antiguos no tienen un enunciado **while**, a veces se necesita un enunciado GOTO, como se indica en el ejemplo de la figura 10.2. Las construcciones EXIT y CYCLE del FORTRAN 90 eliminan la necesidad del GOTO.

GOTO computado. Relacionado con el enunciado IF aritmético, el GOTO computado permite que el control se transfiera entre un conjunto de etiquetas de enunciado posibles, de acuerdo con el valor de un argumento entero.:

$$\text{GOTO } (n_1, n_2, n_3, ..., n_m), \text{I}$$

transfiere el control a la etiqueta n_i bajo el supuesto de que el valor i de la variable I está entre 1 y m. Es una forma primitiva sustituida en la actualidad por el enunciado **case**.

GOTO asignado. Esta es una construcción verdaderamente arcaica.

```
ASSIGN 21 TO K
GOTO K
```

Se asigna una etiqueta de enunciado a K en primera instancia, y la segunda transfiere el control a esa etiqueta. Puesto que el enunciado ASSIGN podría estar en cualquier parte del programa, es imposible entender los programas que usan muchos enunciados GOTO asignados. Evítese esto a cualquier costo. No hay necesidad de usar este enunciado.

Entrada y salida

Se manejan archivos tanto secuenciales como de acceso directo y se suministra un conjunto extenso de operaciones de entrada y salida. Se usan nueve enunciados para operaciones de entrada y salida: READ, WRITE y PRINT especifican la transferencia misma de datos, OPEN, CLOSE e INQUIRE permiten fijar o consultar la situación, método de acceso y otras propiedades de un archivo; y BACKSPACE, REWIND y ENDFILE permiten ubicar el apuntador de posición de archivo.

Un 'textfile' (serie de caracteres) en FORTRAN se conoce como un archivo *con formato*; otros archivos son *sin formato*. Los enunciados READ, WRITE y PRINT convierten los valores de datos de representaciones de almacenamiento internas a forma de caracteres durante la transferencia a archivos con formato. La transferencia de datos a un archivo sin formato deja los datos en su representación interna. Un READ, WRITE o PRINT a un archivo con formato puede ser dirigido por lista, lo que significa que no se proporciona una especificación explícita de formato; en su lugar, se usa un formato implícito definido por el sistema, como en el ejemplo de la figura 10.2. Alternativamente, el programador puede proporcionar un formato explícito:

etiqueta FORMAT (*serie_de_especificaciones_de_FORMAT*)

y la *etiqueta* se designa en el enunciado READ, WRITE o PRINT. Por ejemplo, para leer una serie de enteros con formato de ocho por renglón en campos de cinco caracteres, los enunciados READ y FORMAT podrían ser:

```
      READ 200, N, (M(K), K=1,7)
200   FORMAT (8I5)
```

Esto también se puede especificar colocando la especificación de formato directamente en el enunciado READ:

```
READ "(8I5)" N, (M(K), K=1,7)
```

El enunciado READ especifica que el primer entero se lee a la variable N, y los siete siguientes a los componentes 1 — 7 del vector M. El enunciado FORMAT especifica formato "I" (entero), cinco dígitos por entero, repetido ocho veces a través del renglón que comienza en la posición del primer carácter. Se suministra un gran número de especificaciones de formato posibles para las diversas clases de datos numéricos y de caracteres. Algunas de las más útiles son (w = tamaño del campo y d = número de dígitos decimales):

Campo	Tipo de datos	Ejemplo de uso
Iw	Entero	I5 = 12345
Fw.d	Fijo (real) 1)	F6.2 = 123.56
Ew.d	Exponencial (real) 1)	E10.2 = -12.34E+02 = 1234
wX	Salto	Salta w posiciones de caracteres
Aw	Carácter	A6 = abcdef
"literal"	Constante	"abcdef" se imprime como esos 6 caracteres

En general resulta seguro colocar los datos justificados a la derecha en el campo. Los caracteres iniciales en blanco se interpretan como 0 al leer datos numéricos.

Cada enunciado READ o WRITE da inicio a un nuevo renglón de entrada o salida. Para cada elemento de la lista de entradas o salidas, el elemento FORMAT siguiente se procesa hasta que se alcanza el final de la lista de datos. Si hay más elementos de datos que elementos FORMAT, entonces se inicia de nuevo el enunciado FORMAT sobre un nuevo renglón de datos.

Para especificar un manejador de excepciones para una condición de final de archivo, o un error durante una operación de entrada-salida, se suministra un número de enunciado en el enunciado READ o WRITE que designa un enunciado en el subprograma que está ejecutando el READ o WRITE al cual se deberá transferir el control si se plantea la excepción. Por ejemplo:

```
READ (UNIT=2, FMT=200, END=50, ERR=90) N, (M(K),K=1,7)
```

es la forma ampliada del READ anterior que especifica una transferencia al enunciado número 50 en una condición de final de archivo y al enunciado número 90 cuando hay un error. La designación *UNIT* = 2 especifica que el archivo se puede encontrar en el dispositivo de entrada número 2.

Las especificaciones de formato que se usan para controlar entradas y salidas se tratan como cadenas de caracteres. Durante la traducción se guardan en forma de cadenas de caracteres y son interpretadas dinámicamente por los procedimientos READ, WRITE y PRINT según se requiere durante la ejecución. Una especificación de formato (cadena de caracteres) se puede crear o leer e introducir dinámicamente durante la ejecución del programa y usarse posteriormente para controlar conversiones de entrada-salida. El resultado es un sistema flexible y poderoso de entrada-salida.

10.1.6 Subprogramas y gestión de almacenamiento

Se suministran tres tipos de subprogramas. Los subprogramas de *función* permiten la devolución de un solo valor de tipo numérico, lógico o de cadena de caracteres. Una *subrutina* devuelve resultados a través de cambios en sus parámetros o a través de variables no locales en bloques COMMON. El encabezamiento sólo proporciona los nombres de parámetros formales, por ejemplo:

```
SUBROUTINE SUB1(A, B, C)
```

y los tipos y otros atributos de los parámetros se declaran por separado dentro de la subrutina. Las funciones se definen de manera similar:

```
REAL FUNCTION MISUB(A, B, C)
```

lo cual define una función que devuelve un valor real.

La ejecución de un subprograma (subrutina o función) comienza ordinariamente con el primer enunciado. Sin embargo, se puede usar una declaración ENTRY para suministrar otro punto donde el programa puede iniciar su ejecución cuando se le llama usando un nombre distinto. Por ejemplo, en la subrutina *SUB*1, cuyo encabezamiento se ha indicado, se puede colocar una declaración ENTRY

```
ENTRY SUB2(X, Y)
```

en otro punto del cuerpo, entre dos enunciados. Una llamada, *CALL SUB*2(...), invoca luego la ejecución de *SUB*1 en el punto de la declaración ENTRY (posiblemente con diferente número y tipo de parámetros). El efecto es como si *SUB*1 y *SUB*2 fueran dos subprogramas independientes que comparten porciones de sus cuerpos.

Una función cuyo valor se puede calcular en una sola expresión aritmética o lógica se define como una *función de enunciado*, local para un subprograma particular, y colócase antes del primer enunciado ejecutable de ese subprograma:

```
FN(X, Y) = SIN(X)**2 - COS(Y)**2
```

Las reglas de tipificación implícitas para nombres de variables se aplican para determinar el tipo de ambos parámetros formales y el tipo del resultado que devuelve la función (o los tipos se pueden declarar explícitamente en las declaraciones del programa).

Las funciones de enunciado se pueden llamar como funciones ordinarias dentro de expresiones en el cuerpo del programa en el cual están definidas. Sirven principalmente como una taquigrafía en la escritura de expresiones que ocurren en forma repetida en asignaciones. Puesto que la definición de función de enunciado es local para el subprograma en el cual se usa, cada llamada de función se reemplaza ordinariamente por el cuerpo de la función durante la traducción, y no es necesario un tiempo de procesamiento adicional de llamada y regreso de ejecución. El código está en línea dentro del programa ejecutable. Sin embargo, las funciones de enunciado están en la lista de obsolescencia y probablemente serán eliminadas del lenguaje.

Gestión de almacenamiento

Las referencias en FORTRAN son locales o globales; no se cuenta con providencias para niveles intermedios de referencias no locales antes del FORTRAN 90.

Ambientes locales de referencia. El ambiente local de un subprograma se compone de las variables y arreglos declarados al principio de ese subprograma. Como se implementa FORTRAN ordinariamente, el ambiente local se conserva entre llamadas, porque se asigna almacenamiento estáticamente al registro de activación como parte del segmento de código. Sin embargo, la definición del lenguaje no requiere la retención de ambientes locales a menos que el programador incluya explícitamente el enunciado "SAVE" dentro del subprograma. SAVE indica que el ambiente local completo se debe conservar entre llamadas. Alternativamente, se pueden guardar sólo variables especificadas, por ejemplo, *SAVE A, X* indica que sólo se deben conservar las variables *A* y *X*.

Ambientes comunes. Si se van a compartir variables simples o arreglos entre subprogramas, se deben declarar en forma explícita como parte del ambiente global de referencia. Este ambiente global no se establece en términos de variables y arreglos individuales, sino más bien en términos de *conjuntos de variables y arreglos*, que se conocen como *bloques* COMMON. Un bloque COMMON es un bloque de almacenamiento con nombre y puede contener los valores de cualquier número de variables simples y arreglos (véase la sección 10.1.7).

FORTRAN 90 introduce el concepto de procedimientos anidados con el enunciado CONTAINS:

```
SUBROUTINE SUBRUTINA_EXTERIOR
...
CONTAINS
   SUBROUTINE INTERIOR.1
      ...
   SUBROUTINE INTERIOR.2
      ...
   SUBROUTINE INTERIOR.3
      ...
```

Se pueden asignar referencias a las subrutinas *INTERIOR.*1, *INTERIOR.*2 e *INTERIOR.*3 en *SUBRUTINA_EXTERIOR*, en gran medida como en Pascal.

El FORTRAN 90 también incluye el concepto de procedimientos recursivos:

```
RECURSIVE FUNCTION FACTORIAL(X) RESULT(RESULT_VAL)
```

es la cabecera de un procedimiento recursivo de este tipo. La cláusula RESULT, que proporciona la variable que es devuelta desde el procedimiento, es necesaria en operaciones recursivas, puesto que la convención usual en FORTRAN de asignar un valor a una variable local con el nombre del procedimiento sería ambigua en este caso. Es decir, un enunciado como:

```
FACTORIAL = X * FACTORIAL(X-1)
```

podría ser ambiguo. ¿Está el término *FACTORIAL* a la derecha del valor local de la función factorial, o es una llamada recursiva a la función? No queda claro sin la cláusula RESULT en la cabecera.

Parámetros de subprograma. La transmisión de parámetros es, de manera uniforme, por referencia. Los parámetros reales pueden ser variables simples, literales, nombres de arreglos, variables subindizadas, nombres de subprogramas, o expresiones aritméticas o lógicas.

Los resultados de subprogramas de función se transmiten por asignación dentro del subprograma al nombre del subprograma. El nombre del subprograma actúa como una variable local dentro del subprograma, y la última asignación hecha antes del regreso al programa de llamada es el valor de regreso. Las funciones pueden devolver sólo números individuales, cadenas de caracteres o valores lógicos como resultados.

10.1.7 Abstracción y encapsulamiento

Los subprogramas constituyen el único mecanismo de abstracción en FORTRAN 77, aunque el FORTRAN 90 sí agrega módulos y tipos de datos. Se pueden usar bloques COMMON para aislar datos globales para sólo unos pocos subprogramas que necesitan esos datos, pero la sintaxis de bloques COMMON es muy propensa a errores. Por ejemplo:

```
COMMON/BLK/X,Y,K(25)
```

al aparecer en un subprograma declara un bloque COMMON llamado *BLK*, el cual contiene las variables *X* y *Y* y el vector *K*. Si se supone que *X* y *Y* son de tipo real y *K* es de tipo entero, el bloque *BLK* ocupa 27 unidades de almacenamiento en sucesión en la memoria. Si dos subprogramas *SUB*1 y *SUB*2 contienen ambos la declaración COMMON anterior, entonces el bloque *BLK* es accesible para ambos. Cualquier otro subprograma puede obtener acceso a este mismo bloque COMMON por inclusión de la misma declaración COMMON en la definición del subprograma.

El efecto de la declaración COMMON es permitir que el ambiente global de referencia se divida en *bloques* para que cada subprograma no necesite tener acceso a todo el ambiente global. La declaración COMMON permite un procesamiento eficiente en tiempo de ejecución porque cualquier referencia a una variable global en un subprograma se puede compilar de inmediato a un cómputo de dirección base más desplazamiento en tiempo de ejecución, donde la dirección base es la dirección del principio del bloque COMMON.

El enunciado COMMON se usa casi siempre en una forma idéntica en cada subprograma que tiene acceso a un bloque COMMON dado. Sin embargo, la estructura permite también otras posibilidades. Obsérvese que es sólo el *nombre* de un bloque COMMON el que es de hecho un identificador global; los nombres de variables y arreglos de la lista de enunciados COMMON son *locales* al subprograma en el cual aparece el enunciado; es decir, es sólo el bloque de almacenamiento que contiene los valores de los identificadores lo que se está compartiendo en efecto, no los identificadores mismos. Por tanto, es posible tener distintos identificadores y organizar el bloque COMMON de manera diferente en cada subprograma que lo utiliza. Por ejemplo, si el subprograma SUB1 contuviera el enunciado COMMON antes citado, SUB2 podría contener:

```
COMMON/BLK/U,V,I(5),M(4,5)
```

Este bloque COMMON también ocupa 27 unidades de almacenamiento, y el tipo asignado a cada elemento del bloque es el mismo en cada caso. El segundo enunciado COMMON asigna un conjunto completamente distinto de identificadores y estructuras de arreglos a las 27 unidades de almacenamiento, pero una asignación de esta clase sólo es válida en tanto se conserve la longitud total y la estructura de tipos del bloque.

10.1.8 Evaluación del lenguaje

El FORTRAN tiene un sorprendente poder de permanencia. Fue declarado como obsoleto por muchas personas en los años sesenta, pero el FORTRAN 90 sigue entre nosotros. No es tan popular como lo fue alguna vez, pero es todavía el lenguaje de elección para muchas personas que tienen que ver con el cómputo científico. Las técnicas de compilación para evaluación de expresiones aritméticas, cálculos de matrices ralas y problemas de procesamiento de arreglos grandes han llegado a ser bastante buenas, y para los cálculos numéricos que se apoyan en el cómputo de derivadas, integración numérica, dinámica de fluidos y obtención de raíces de ecuaciones complejas, el FORTRAN es tan eficiente como cualquier lenguaje.

Los renglones individuales de un programa en FORTRAN son por lo común fáciles de entender. Sin embargo, la estructura global de los programas tiende a ser más bien opaca en razón del uso abundante de etiquetas de enunciado y GOTOs en los mecanismos de control de secuencia. Por tanto, suele ser difícil ver el flujo global del control en un programa en FORTRAN. También el uso de identificadores nemotécnicos se dificulta por una restricción a identificadores de seis caracteres en los FORTRAN antiguos. La adición de funciones recursivas y arreglos dinámicos en el FORTRAN 90 proporciona a este lenguaje un poderoso modelo de ejecución.

Conforme el lenguaje evoluciona, sin embargo, ciertas limitaciones más antiguas se están volviendo menos importantes. Las etiquetas de enunciados ya no se necesitan conforme se introducen estructuras de control de multienunciados anidados, y son por completo innecesarias en el FORTRAN 90. Las definiciones de tipos se basan en una semántica como la del Pascal, y se ha introducido almacenamiento dinámico y variables apuntador. Los nombres pueden ser más largos y más nemotécnicos. Sin embargo, el lenguaje tiene todavía un aspecto acartonado que es un remanente de sus inicios de "mazo de cartas" de procesamiento por lotes.

10.2 C

El C es un lenguaje desarrollado en 1972 por Dennis Ritchie y Ken Thompson de los Bell Telephone Laboratories de AT&T. Está relacionado con el ALGOL y el Pascal en cuanto a estilo, y con la inclusión de atributos del PL/I. Aunque es un lenguaje de programación para usos múltiples, su sintaxis compacta y sus características de ejecución eficiente lo han vuelto popular como lenguaje de programación de sistemas.

10.2.1 Historia

A finales de la década de 1960, los Bell Telephone Laboratories de AT&T se separaron del proyecto del MIT y la GE encaminado a desarrollar el sistema operativo Multics; sin embargo, el desarrollo de un sistema útil continuaba como objetivo para Ken Thompson. Así se inició UNIX, cuyo nombre es un juego de palabras con el nombre Multics. Multics estaba programado en PL/I, y existía un fuerte deseo de construir este lenguaje usando también un lenguaje de alto nivel, aunque se consideraba que el PL/I era demasiado engorroso. Con cierta experiencia previa con el lenguaje de sistemas BCPL (cuyo nivel era demasiado bajo y carecía de todo apoyo en tiempo de ejecución), Thompson proyectó un lenguaje llamado B como un subconjunto mínimo del BCPL para programación de sistemas ("BCPL comprimido a bytes de 8K de memoria [de PDP-7]" [RITCHIE 1993]). En la actualidad es difícil darse cuenta que restricción tan fuerte era un tamaño de memoria limitado hace sólo 20 años.

En 1970, el proyecto UNIX adquirió una PDP-11 con su enorme memoria de 24K. En este momento, la pequeña pero creciente comunidad embriónica UNIX se sintió limitada por las restricciones de B. Se agregaron tipos, definiciones de estructuras y operadores adicionales, y el nuevo lenguaje se dio a conocer como C.

Aunque es un lenguaje de programación para usos generales, se le ha asociado estrechamente con actividades de programación de sistemas. Se usó por primera vez para escribir el núcleo del sistema operativo UNIX y desde entonces ha estado fuertemente ligado a las implementaciones de UNIX, aunque existen versiones de C en casi todos los sistemas de cómputo actuales.

Durante los años setenta, C fue principalmente una curiosidad universitaria debido a la fascinación de las universidades con UNIX. Sin embargo, en los años ochenta, conforme comenzaron a aparecer versiones comerciales de este sistema operativo, la exposición que sufrió C hizo crecer su popularidad. En 1982, un grupo de trabajo de ANSI comenzó a desarrollar un estándar para el lenguaje, el cual se produjo finalmente en 1989 [ANSI 1989] y fue aceptado como un estándar internacional [ISO/IEC 9899) en 1990.

En la actualidad es probable que los programadores en C conformen el segmento en más rápido crecimiento del mundo de la programación. Con su derivado C++, C y C++ constituyen influencias considerables en la programación actual. Aunque el Pascal fue el lenguaje universitario de elección durante gran parte de la década de 1980, hoy día el C ha sustituido al Pascal en muchos cursos de iniciación a la programación.

10.2.2 Hola Mundo

El C se compila usando tecnología convencional de compiladores. Se utiliza un editor para crear archivos fuente y luego un compilador crea el programa objeto ejecutable, el cual se ejecuta por separado. La figura 10.3 es nuestro programa trivial "Hola Mundo" en C. `cc`, de *Compilador de C,* es la designación casi universal del traductor de C.

```
%edit trivial.c
      main()
      {int i;
         i=2;
         if(i >= 2) printit();
      }
      printit()
         {printf("Hola Mundo\n");}
%cc trivial.c
%execute
Hola Mundo
```

Figura 10.3. "Hola Mundo" en C

10.2.3 Breve perspectiva del lenguaje

Cuando se analiza el C, casi siempre se considera el "concepto C" y no sólo el lenguajedefinido por una gramática específica. La "programación en C" casi siempre se compone de lo siguiente:

- *El lenguaje C.* Se trata de un lenguaje relativamente pequeño con un número limitado de estructuras de control y características. (Recuérdense sus raíces como un compilador mínimo que se ejecutaba en una PDP 7 y más tarde en una PDP 11 pequeña.)
- *El preprocesador de C.* Puesto que casi todos los compiladores de C incluyen enunciados de preprocesador #, la mayoría de los usuarios no se dan cuenta de que en realidad éstos no forman parte del lenguaje C.
- *Los supuestos de interfaz de C.* Se ha desarrollado un conjunto de convenciones respecto al uso de las características de C. Por ejemplo, se supone que las definiciones de interfaz entre módulos están definidas en un archivo ".h" (*header*; cabecera) apropiado. El enunciado:

  #include "mifcn.h"

  utiliza tanto el preprocesador de C como esta tercera convención al especificar la interfaz con el módulo *mifcn*.
- *La biblioteca de C.* Muchas funciones, *printf*, *getchar*, *malloc*, *fork*, *exec*, etc., han sido escritas con interfaces de lenguaje C, aunque no son oficialmente parte de la definición de ese lenguaje . Sin embargo, el C de ANSI incluye estas funciones como funciones de biblioteca requeridas para conformar compiladores. La inclusión de una biblioteca grande permite tener un lenguaje núcleo relativamente pequeño, el cual se puede mejorar con estas funciones de biblioteca.

Un módulo de C consiste en declaraciones globales y una sucesión de funciones. Se cargan juntos múltiples módulos para formar un programa ejecutable. Cada función puede invocar otras funciones y tener acceso a datos locales o globales para esa función.

Con esta estructura, el almacenamiento de datos es muy simple. Cada función tiene un registro de activación local, el cual es dinámico y da cabida a recursión, y cada función tiene acceso a datos globales. No hay una auténtica estructura de bloques. Por tanto, los registros de activación no necesitan vínculos estáticos, sólo vínculos dinámicos. Cada elemento de datos tiene una implementación eficiente. Por ejemplo, los arreglos con múltiples dimensiones se construyen a partir de arreglos unidimensionales, y éstos tienen índices que comienzan con 0, lo cual evita la necesidad de descriptores. La dirección inicial del arreglo coincide con el origen virtual, lo cual evita todos los cálculos de arreglos complejos.

El lenguaje C tiene apuntadores, y existe una equivalencia entre arreglos y apuntadores, lo que permite que los programas utilicen el método de acceso más apropiado. Las cadenas se implementan como arreglos de caracteres. Esta implementación es completamente transparente, de modo que se puede tener acceso a las cadenas como cadenas, como arreglos o, como se acaba de señalar, como apuntadores a caracteres individuales.

El C tiene un conjunto grande de operadores aritméticos que dan origen a programas muy eficientes y a veces muy concisos. Este lenguaje tiene un conjunto completo de estructuras de control con una semántica muy flexible, lo que en ocasiones da cabida a usos extraños.

C tiene también una vía flexible para definición de tipos; sin embargo, se puede afirmar (correctamente) que C es a la vez de tipos fuertes y no de tipos fuertes. La razón de esto es que casi todos los elementos de datos son subtipos de enteros. Aunque el traductor puede encontrar errores de tipificación, como casi todos los elementos de datos son enteros en último término, se pasan por alto muchos errores.

El C siempre ha estado estrechamente ligado a la funcionalidad de sistemas operativos. En UNIX, las funciones del sistema operativo (por ejemplo, la función *malloc* para asignar almacenamiento dinámico) se especifican como llamadas de función de lenguaje C. Por convención, éstas se especifican en archivos ".h" de sistema. Así, por ejemplo, para llamar *malloc* desde un programa en C, el programa incluiría:

```
#include <malloc.h>
```

al principio del programa, y simplemente se usa *malloc*(*TamañoDeAlmacenamiento*) cuando se necesita en el mismo. Cuando un programa de este tipo se compila y se carga, las definiciones de función apropiadas (por ejemplo, *malloc*) se incluyen desde la biblioteca de C.

El compilador de C se ejecuta invocando primero el preprocesador. Los mandatos como *#define* y *#include* se evalúan primero, y luego el traductor de C compila el programa completo.

Los enunciados de entrada-salida siguen el patrón del concepto FORMAT de FORTRAN, pero se hacen cargo de la ejecución de programas interactivos de mejor manera que los enunciados READ y WRITE de FORTRAN. Casi todas las funciones útiles están definidas en el archivo de sistema stdio.h, el cual se debe incluir en todos los programas en C. Esto proporciona una forma fácil de ampliar un lenguaje; se escribe un conjunto de funciones para agregar nueva funcionalidad al lenguaje.

Los comentarios (cualquier texto acotado por /* ... */) pueden aparecer en cualquier punto donde se pueda usar un espacio en blanco en el programa. Puesto que el texto es de formato libre en los archivos fuente, por lo general no se necesitan líneas de continuación; sin embar-

```
1     #include <stdio.h>
2     const int tammáx=9
3     main()
4        {int a[tammáx];
5        int j,k;
6        while( (k=convert(getchar())) != 0) {
7           for(j=0; j<k; j++) a[j] = convert(getchar());
8           for(j=0; j<k; j++) printf("%d ", a[j]);
9           printf(«; SUMA= %d\n», addition(a,k));
10          while(getchar() != '\n');
11          } }
12    /* Subprograma de conversión de funciones */
13    int convert(char ch)
14       {return ch-'0';}
15    /* Subprograma de adición de funciones */
16    int addition(v, n)
17       int v[ ], n;
18       {int sum,j;
19       sum=0;
20       for(j=0; j<n; j++) sum=sum+v[j];
21       return sum;
22       }
```

Figura 10.4. Ejemplo en C para sumar un arreglo.

go los enunciados de preprocesador de macros se definen como un renglón de entrada. Por tanto, un enunciado de preprocesador de macros emplea el símbolo \ para indicar que el enunciado de preprocesador continúa en el renglón siguiente.

Ejemplo anotado

El programa de la figura 10.4 es similar al ejemplo en FORTRAN de leer y sumar las entradas en un arreglo. Sin embargo, este ejemplo muestra algunas distinciones importantes de C. Aunque el lenguaje es eficiente, muchos de los detalles de "mantenimiento de datos" se dejan al programador. Por consiguiente, es mayor la estructura adicional necesaria para comenzar a escribir un programa en C. Sin embargo, una vez que se ha escrito este código de estructura, se facilita la escritura del resto del programa. Con frecuencia, los programadores tienen su propia biblioteca de esta clase de módulos, los cuales son necesarios para crear las operaciones básicas que se requieren para poner un programa en ejecución.

En este ejemplo, una entrada de:

```
41234
0
```

producirá una salida de:

```
1 2 3 4 ; SUMA= 10
```

Renglón 1. La biblioteca normal de entrada-salida stdio.h se debe usar en este programa. Se trata de un enunciado de preprocesador que es expandido antes de que se inicie la traducción. El contenido del archivo `stdio.h` se incluye como parte del programa en este punto.

Renglón 2. `tammáx` se define como una constante entera con valor 9. También se podría haber usado un enunciado de preprocesador con resultados similares:

```
#define tammáx 9
```

Renglón 3. La ejecución principia aquí. Alguna función debe tener el nombre `main`.

Renglón 4. El arreglo de enteros `a` se declara con subíndices que van de 0 a 8.

Renglón 5. `j` y `k` son variables enteras locales para `main`.

Renglón 6. `getchar()` es una función definida en `stdio.h`, la cual lee e introduce el carácter siguiente de la entrada. Puesto que las variables char son subtipos de los enteros, se lee e introduce el valor entero del carácter. Por ejemplo, en sistemas que emplean caracteres ASCII, el dígito 2 es el carácter con el valor en bits 0110010, que es 62 en octal o 50 en decimal. Sin embargo, el programa desea el valor numérico 2, y esto es lo que hace la función `convert` (renglones 13-14).

El valor que se lee e introduce se asigna a `k`. Obsérvese que la asignación es aquí sólo un operador, y la expresión `k=convert(getchar())` asigna el número de entrada siguiente a `k` y devuelve eso como valor de la expresión. Si el valor no es 0, entonces se ejecutan los enunciados siguientes (renglones 7-11). Por consiguiente, un valor de 0 termina la iteración y el programa "cayéndose del final" del procedimiento `main`.

Renglón 7. El enunciado `for` lee e introduce valores sucesivos del arreglo (usando la función `convert` y la función stdio `getchar`). El primer argumento del `for` se ejecuta al entrar al enunciado, y es comúnmente el valor inicial de la iteración (`j=0`). El segundo argumento es la condición de continuación (hacer iteraciones en tanto `j<k`), mientras que el tercer argumento es la operación al final de la iteración (`j++` o sumar uno a `j`).

Renglón 8. Este enunciado `for` es similar al anterior, con el propósito de imprimir el arreglo. La función `printf` imprime su argumento de cadena de caracteres El `%d` indica que C debe tomar el próximo argumento e imprimirlo en formato de entero en ese punto de la cadena.

Renglón 9. Esta función `printf` es similar a la del renglón 8. Sin embargo, el argumento es el resultado de la llamada de función `addition(a, k)`. El \n es el código en C para que se imprima un indicador de fin de renglón; es decir, el renglón, integrado en el *buffer* de salida de C, se imprime o se exhibe efectivamente en la terminal.

Renglón 10. El programa lee caracteres hasta que se lee el marcador de final de renglón. Este enunciado `while` tiene un cuerpo de enunciado nulo. El siguiente que se lea en el renglón 6 será el primer carácter del próximo renglón de entrada.

Renglón 11. Estas llaves concluyen el bloque que se inicia en el renglón 6 y el cuerpo del programa desde el renglón 4.

Renglón 12. Un comentario. Éstos se pueden colocar en cualquier punto donde se pueda poner un carácter en blanco.

Renglón 13. Procedimiento `convert` de enteros. La colocación del tipo de parámetros en la lista de argumentos es una convención de ANSI en C.

Renglón 14. No es necesario conocer el valor ASCII de los dígitos. Todo lo que se debe saber es que '0' es 0, '1' es uno más que '0' con un valor de 1, '2' es 2 más que '0' con un valor de 2, y así sucesivamente. La expresión `ch-'0'` devuelve el valor de un solo dígito que se requiere.

Renglones 16-17. Este es el estilo antiguo de C para parámetros. Los nombres se colocan en la lista de argumentos y sus tipos aparecen a continuación. Adviértase que no se especifican los límites de arreglo. Puesto que sólo existen arreglos unidimensionales y no se necesitan descriptores, el tamaño del arreglo no proporciona información necesaria para el compilador. De hecho, esto se compila como si el argumento estuviera escrito como $*v$, donde v es un apuntador.

Renglones 18-19. `sum` y `j` son variables locales en la función `addition`. `sum` se inicializa a 0.

Renglón 20. Otra iteración `for` similar a los renglones 7 y 8.

Renglón 21. Devuelve `sum` como valor de la función `addition`.

Renglón 22. Fin de la función `addition`.

10.2.4 Objetos de datos

C tiene tipos fuertes, aunque son muy pocos, de modo que esta tipificación fuerte ayuda poco. Los datos pueden ser enteros, tipos enumerados o 'float' (de punto flotante). Los datos estructurados pueden ser arreglos, cadenas (una forma de arreglos) o registros como 'struct' o 'union'.

Tipos de datos primitivos

Variables y constantes. Los identificadores en C son cualquier serie de letras, dígitos y el carácter de subrayado (_), y no deben comenzar con un dígito. Los archivos de cabecera ".h" de C suelen utilizar nombres de variables internas que comienzan con una línea de subrayado, así que es preferible evitar esa convención en los programas que uno escribe. Las mayúsculas y minúsculas son importantes en C, abc y ABC son nombres distintos.

Los enteros se pueden especificar en el formato entero usual (por ejemplo, 12, −17) o como constantes octales que contienen un 0 inicial (por ejemplo, 03 = 3 decimal, 011 = 9 decimal, 0100 = 64 decimal). Una literal de caracteres se escribe entre comillas (por ejemplo, '*A*', '*B*', '1'). El carácter \n representa el carácter de final de renglón, y \0 es el símbolo *nulo*, que se requiere en el procesamiento de cadenas. Los códigos mismos de los caracteres ASCII se pueden especificar usando códigos numéricos. Así, \062 es el carácter ASCII 62 (octal), que es lo mismo que '2'.

Se pueden crear valores discretos por medio de *tipos de enumeración*, que son otra forma de datos enteros:

enum *nombredetipo* { *lista de valores* } *name*

como en:

```
enum colores {rojo, amarillo, azul} x, y, z;
x= rojo;
y = azul;
```

Los valores para el tipo enumerado sólo se tienen que detallar una vez. Una segunda declaración para *colores* se puede escribir como:

```
enum colores a, b, c;
```

Las constantes de punto flotante son similares a las convenciones de FORTRAN. Los números tienen ya sea un punto decimal intercalado (por ejemplo, 1.0, 3.456) o un exponente (por ejemplo, 1.2e3 = 1200., .12e–1 = 0.12).

Las constantes de cadena se especifican como caracteres encerrados entre comillas (por ejemplo, "abc", "1234"). Sin embargo, las cadenas son en realidad arreglos de caracteres con terminadores nulos. Por consiguiente, la cadena "123456" es de hecho la secuencia 123456\0 y se puede procesar como si estuviera declarada como una de las siguientes:

```
char vardearreglo[] = "123456"; /* vardearreglo de longitud 7 */
char *aptdearreglo = "123456"; /* aptdearreglo apunta a los caracteres */
```

Tipos de datos numéricos. Los enteros tienen diversas variantes. El tipo *short* (corto) es un entero que utiliza en efecto el número mínimo de bits para la implementación dada. Para muchas implementaciones de computadora personal, esto significa ordinariamente un entero de 16 bits con signo que tiene un valor máximo de 32 767. También puede ser más grande —como un entero de 32 bits—.. El tipo *long* (largo) es la implementación de hardware más larga para enteros en ese hardware particular. *int* puede ser igual a *short*, puede ser igual a *long*, o puede tener algún valor intermedio entre esas longitudes.

Como lo demostró el ejemplo anotado, el tipo *char* es también un subtipo del entero, en este caso un valor de 1 byte. Por tanto, cualesquier caracteres que se lean e introduzcan se pueden manipular como cualquier otro valor entero.

No existen datos booleanos o lógicos; los enteros sirven para este propósito. *TRUE* (cierto) se define como cualquier valor entero diferente de cero y *FALSE* (falso) como el valor cero. Se pueden usar operadores de bits para fijar y restablecer diversos bits en los enteros. Aunque se pueden usar varios bits en una sola palabra para guardar distintos valores booleanos (por ejemplo, el número binario 101 = 5 decimal podría significar "A es cierto, B es falso, C es cierto"), suele ser mejor utilizar variables *int* para este propósito.

Las constantes se pueden definir usando ya sea el enunciado de proprocesador #define, como se ha indicado, o la declaración *const*:

const *nombre de tipo* = *valor*

Datos apuntador. Los apuntadores o punteros pueden señalar a cualquier dato. **p* indica que *p* es un apuntador a un objeto. Por ejemplo, los enunciados:

```
int *p;
p = (int *) malloc(sizeof(int));
p = 1
```

hacen lo siguiente:

1. Asignan un apuntador *p* que señala a un entero.
2. Asignan memoria suficiente para guardar un entero y guardan el valor l de esa memoria como valor r de *p*.
3. Usando el valor r de *p*, guardan el entero 1 en el espacio de memoria asignado a *p* y al cual apunta *p*.

Tipos de datos estructurados

Arreglos. Los arreglos son unidimensionales con índices que comienzan con 0: por ejemplo, *int a*[3]. Se pueden definir arreglos multidimensionales de cualquier número de dimensiones:

```
int a[10][5]
```

Estos arreglos se guardan en un orden *por filas*. El efecto de esto es que el arreglo se guarda como si estuviera declarado en esta forma:

```
cosa a[10]
```

donde el tipo *cosa* es un arreglo de enteros de longitud 5.

Las cadenas son arreglos de caracteres, como ya se ha descrito. Las funciones de cadena emplean por lo general cadenas de terminación nula; por ejemplo, arreglos de caracteres terminados con el carácter nulo: "*abcdef*" = *abcdef*\0.

Tipos definidos por el usuario

Como lenguaje, C puede crear registros de datos estructurados llamados **struct**. La sintaxis es similar a las constantes enumeradas anteriormente indicadas:

```
struct ClaseMéxico {int tamaño; char profesor[20];} CMSC330, CMSC630;
```

define una estructura *ClaseMéxico* que contiene un componente *tamaño* y un componente *profesor* que es un arreglo de 20 caracteres, con dos variables llamadas *CMSC*330 y *CMSC*630 del tipo *ClaseMéxico*.

```
struct ClaseMéxico CMSC430;
```

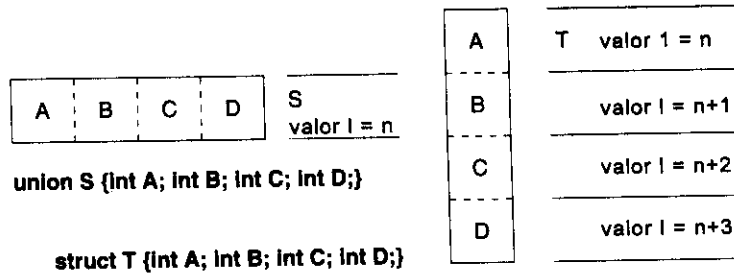


Figura 10.5. Almacenamiento para la union S y la struct T.

define una nueva variable *CMSC*430 como otra *ClaseMéxico*, referente a la misma definición de *struct*.

El acceso a miembros de la estructura se efectúa a través de una notación de nombre calificado (con punto). *CMSC*330.*tamaño* es el componente de tamaño de entero de esa estructura y el nombre calificado *CMSC*330.*profesor* es el componente de profesor de la misma.

La declaración **struct** crea un tipo nuevo. Paradójicamente, la declaración **typedef** es en realidad una sustitución de nombre y no una nueva declaración de tipo:

```
typedef int nuevoent;
```

define *nuevoent* como lo mismo que *int*. Se usa por lo general en definiciones de **struct**:

```
typedef struct ClaseNueva { ... } ClaseMéxico;
ClaseMéxico A;
ClaseMéxico B;
```

define que *ClaseMéxico* es la definición de **struct** *ClaseNueva*, y la definición de tipo define las variables *A* y *B* como pertenecientes al tipo *struct ClaseNueva*.

Union. Una **union** es una definición de tipo, la cual es sintácticamente como la definición de tipo **struct**, pero desde el punto de vista semántico es como el registro de variante de Pascal de la página 165. La diferencia entre un tipo **unión** y uno **struct** se muestra en la figura 10.5. Con la *union S*, cada componente de *S* tiene un valor l, el cual es igual al valor l de *S*. Cada componente ocupa la misma localidad de almacenamiento. En la *struct T*, el valor l de *A* es igual que el valor l de *T*, pero los otros son todos diferentes. Cada componente tiene su propia localidad de almacenamiento.

Representación de almacenamiento

Los objetos en C emplean básicamente la representación de hardware de sus tipos. Los enteros, 'float' y caracteres se guardan en su formato natural, y los arreglos no necesitan descriptores. Los apuntadores son simplemente el valor l del objeto al que señalan. Es poco el tiempo de procesamiento adicional para tener acceso a estos objetos.

Los apuntadores y los arreglos están estrechamente relacionados. También se puede tener acceso al arreglo:

```
int A[10]
```

como una variable apuntador:

```
int *aptdearreglo;
aptdearreglo = &A[0]; /* aptdearreglo señala a A[0] */
```

La expresión *((*aptdearreglo*) + (*i*)) es lo mismo que *a*[*i*].

Si *q* es una **struct**, como:

```
struct q {int a; int b; }
```

entonces la secuencia:

```
struct q *p
p = (struct q *) malloc(sizeof(struct q));
```

como en el ejemplo anterior, asigna memoria para la **struct** *q* y guarda su dirección en *p*. El operador → (escrito como –>) se usa entonces para tener acceso a componentes de esta **struct**. Se hace referencia al componente *a* como $p \rightarrow a$ y al componente *b* como $p \rightarrow b$. Adviértase que $p \rightarrow a$ es *exactamente* lo mismo que (*p).*a*.

Los datos *bit* binarios son también una forma del tipo entero. Los datos de bits pueden ser unsigned (sin signo) (enteros), signed (con signo) (enteros) o *int* (que quiere decir con signo). Los bits con signo y sin signo pueden tener una longitud especificada, como en:

```
unsigned CampoDeDatos:7
```

que describe *CampoDeDatos* como un entero sin signo de 7 bits con valores desde 0 hasta $2^7 - 1 = 127$. Si *CampoDeDatos* fuera *signed*, tendría un ámbito de valores desde –64 hasta +63, puesto que uno de los bits representaría un bit de signo.

Al asignar almacenamiento para datos enteros, los campos de bits no se deben usar simplemente para ahorrar espacio. La **struct** de 16 bits:

```
struct AlmacénDeBits {unsigned a:7; unsigned b:9} c;
```

guarda *c.a* con valores que van de 0 a 127 en 7 bits y *c.b* con valores que van de 0 a 511 en 9 bits. Sin embargo, el acceso a *c.a* y *c.b* requiere desplazamiento complejo de bits en una palabra. El código adicional que se requiere puede eliminar cualquier ahorro potencial en espacio para datos. Una declaración como:

```
struct AlmacénDeDatos {int a; int b} c;
```

es mucho más eficiente, aun cuando requiere unos pocos bits más para guardarse. Estas estructuras de bits son de mayor utilidad cuando se está sobreponiendo una plantilla de patrones de bits sobre una estructura de datos ya existente. Por ejemplo, si se apunta a un número de punto flotante, se pueden usar campos de bits como una forma de tener acceso a los componentes individuales de signo, exponente y mantisa del número.

Inicialización. Se puede inicializar cualquier variable asignada estáticamente, como en:

```
int i= 12;
```

Los valores dinámicos de un registro de activación no se pueden inicializar. Si se va a inicializar un arreglo, se especifica una lista de elementos:

```
int a[4]= (1,2,3,4); char cadena[4] = "abc";
```

Este último ejemplo inicializa el arreglo *cadena* a *abc*\0. Se debe reservar espacio en el arreglo para el carácter nulo final.

10.2.5 Control de secuencia

Expresiones

El lenguaje C utiliza la notación infija usual para expresiones. Uno de los puntos fuertes del C es el número de operadores que pueden manipular datos numéricos. (También es una debilidad, puesto que ahora se dispone de muchas formas de llevar a cabo operaciones similares.) El conjunto de operaciones de C y sus niveles de precedencia se presentó ya en la tabla 6.2.

Es importante recordar la distinción entre operaciones lógicas y por bits. El *and* por bits en 5 & 4 dará por resultado un valor de expresión de 4, puesto que aplica *and* a los bits en forma individual:

$$5 \,\&\, 4 = 0101 \,\&\, 0100 = 0100 = 4$$

pero el *and* lógico de la expresión 5 && 4 da por resultado un valor de 1, puesto que opera como sigue:

$$a \,\&\&\, b = \quad \textit{if } a = 0 \textit{ then } 0 \textit{ else if } b = 0 \textit{ then } 0 \textit{ else } 1$$

Los otros operadores lógicos se manejan en forma similar. Siempre devuelven un valor de 0 o 1.

La expresión "if ($a = b$)" asigna b a a y devuelve el valor de a. Se debe tener cuidado si lo que se quiere expresar en realidad es la prueba de igualdad "if (a==b)".

Coerciones. Las coerciones ocurren por lo general si no hay pérdida de información, como de tipo *char* a tipo *int*. Las coerciones se pueden forzar escribiendo una *cast* unaria antes de un valor. Así, a + (*int*)b restringe a b al tipo *int* antes de sumarla a a.

Enunciados

Bloques. La secuencia {*lista de enunciados*} se puede usar siempre que se necesita un enunciado. Además, cualquier bloque puede declarar variables locales:

```
{int i, j; ...}
```

Sin embargo, como se explica en la sección 7.3.4, el almacenamiento para i y j se asignará cuando se haga la asignación de almacenamiento para el procedimiento completo que contiene este bloque.

Enunciados de expresión. Cualquier expresión se puede usar como un enunciado. En particular, el enunciado de asignación $a = b$ es en realidad la expresión de asignación.

Enunciado condicional. El C dispone de las construcciones **if then** e **if then else** usuales:

if then: if(*expresión*) *enunciado*
if then else: if(*expresión*) *enunciado* else *enunciado*

En el caso de enunciados **if** anidados, el enunciado **else** está asociado con el **if** más próximo.

Enunciados iterativos. Existen tres enunciados iterativos en C: **while**, **do** y **for**:

while: while (*expresión*) *enunciado*; significa ejecutar *enunciado* en tanto *expresión* continúe siendo cierta.

do: do *enunciado* while (*expresión*); significa ejecutar *enunciado* y luego poner a prueba la *expresión*. Si es cierta, entonces el **do** se repite. A esto se le suele llamar el **repeat until** (repetir hasta que) en Pascal y en otros lenguajes, excepto que la expresión de prueba continúa la iteración si es cierta.

for: for($expr_1$; $expr_2$; $expr_3$) *enunciado*; es una forma de iteración y se ejecuta como sigue:

1. Si $expr_1$ está presente, se evalúa. Suele ser un enunciado de inicialización como $J = 0$, pero no es necesario que lo sea.
2. Si $expr_2$ está presente, se evalúa. Si el resultado es 0, entonces el **for** termina. Por lo común define la condición de detención de la iteración, como $J < 10$.
3. Se ejecuta *enunciado*.
4. Si $expr_3$ está presente, se evalúa. Por lo común es el incremento, como J++, o suma uno a J.
5. Repetición del proceso con el paso 2.

Enunciado switch. El **switch** (cambiar) es como una ramificación de vías múltiples y se le suele llamar enunciado **case**. Tiene la sintaxis:

```
switch( expresión)
    {case constante1: ListaDeEnunciados1;break;
    case constante2: ListaDeEnunciados2;break;
    case constante3: ...;
    default: ListaDeEnunciadosn;}
```

La *expresión* se evalúa y el control pasa a la etiqueta del enunciado case con el valor constante correcto, o al enunciado por omisión si ninguna etiqueta concuerda.

El **break** es importante aquí y es una auténtica debilidad de C, puesto que un enunciado case fluirá al siguiente si no hay transferencia de control.

Enunciados para transferir el control. Los cuatro enunciados que transfieren el control son **break, continue, goto** y **return**.

break hace que el enunciado envolvente **while, do, for** o **switch** (sintácticamente) más pequeño termine. Es en esencia un enunciado goto al primer enunciado que sigue a este enunciado compuesto envolvente. Como se planteó anteriormente, es importante que el enunciado **switch** evite que un enunciado case fluya al siguiente.

continue transfiere el control al fondo del enunciado envolvente **for, do** o **while** más pequeño. Hace que el programa pase a la próxima iteración de ese enunciado.

goto *etiquetaderamificación* transfiere el control al enunciado que tiene el rótulo *etiquetaderamificación*. Como en FORTRAN, es un enunciado deficiente, y su uso es innecesario y se debe evitar. Los enunciados **break** y **continue** anidados dentro de los enunciados de iteración constituyen todo el mecanismo de transferencia que se requiere.

return devuelve desde un procedimiento. Si el procedimiento fue llamado como función, la sintaxis es: return *expresión*.

Enunciados de preprocesador. define, ifdef, ifndef, include, if, undef y **else** son enunciados de preprocesador que comienzan con el símbolo #, el cual no tiene otra función en C. Para los traductores de C más antiguos, el símbolo # debe ser el primer símbolo del renglón; sin embargo, en muchos traductores actuales, se permite que el # vaya precedido por espacios en blanco.

#define da nombre a una serie de componentes léxicos, como en:

#define *nombre SerieDeComponentesLéxicos*

Se pueden definir constantes, como en:

#define TRUE 1

Adviértase que:

#define TRUE = 1

es incorrecto, puesto que el valor de TRUE sería "= 1", no simplemente "1".

#define también se puede usar para definiciones de macros:

#define *nombre*(Var_1, Var_2,..., Var_n) *SerieDeComponentesLéxicos*

donde las *var* se reemplazan por sus argumentos reales cuando se utilizan.

#define abs(A,B) A<B ? A: B

abs se puede usar entonces en cualquier expresión. *abs*(*varnueva, varantigua*) da por resultado que el traductor compile la expresión *varnueva* < *varantigua*?*varnueva* : *varantigua*.

#include agrega texto al programa desde un archivo:

#include < *nombredearchivo* >
#include " *nombredearchivo* "

El primero agrega *nombredearchivo* desde la biblioteca del sistema, que para sistemas UNIX es comúnmente */usr/include*. El segundo agrega *nombredearchivo* desde el mismo directorio del programa fuente que se está compilando. La convención es que las definiciones de interfaz se guardan en archivos #include ".h" y se copian según se requiere.

#ifdef se usa para verificar si un nombre ha sido definido previamente. Si está definido, entonces se agrega esta serie de enunciados al programa fuente:

#ifdef *NombreDeVariable*
enunciados
#endif

En forma similar, **ifndef** agrega los enunciados si el *NombreDeVariable* no está definido.

#if agrega texto en forma condicional con base en el valor de una expresión constante.

#undef hace que un nombre ya no esté definido: #undef *NombreDeVariable*.

#else se puede usar para iniciar una cláusula else por si fracasa #if, #ifdef o #ifndef.

Entrada y salida

El C no tiene enunciados específicos de entrada o salida. Las entradas y salidas se manejan por medio de un conjunto previamente especificado de funciones que están definidas en el archivo de cabecera *stdio.h*. La entradas y salidas se describen bajo el título de "Funciones normales" en la sección siguiente acerca de subprogramas y gestión de almacenamiento.

10.2.6 Subprogramas y gestión de almacenamiento

Un programa en C tiene la estructura:

Serie de declaraciones globales
Serie de definiciones de función

Una función debe tener el nombre *main* (principal) y es la primera función donde se inicia la ejecución. El C asigna las declaraciones globales en forma estática y luego crea un registro de activación para cada función conforme la misma es llamada.

Ambiente local de referencia. Puesto que no hay anidamiento de funciones, toda variable es local o global. Para cada registro de activación de procedimiento que se crea, no se necesita un vínculo estático a un bloque envolvente. Sólo se requiere un apuntador de vínculo dinámico al registro de activación de la función de llamada para devolver al registro de activación de llamada.

Ambiente común de referencia. Todos los objetos de datos de un módulo definido de manera global para las funciones del módulo son accesibles para todas las funciones del módulo. Los datos globales que se usan en varios módulos deben tener el atributo *extern* (externo) en ellos para que sean visibles dentro de otro módulo:

```
extern int i,j,k;
```

Esto expresa que *i*, *j* y *k* son todas variables globales externas, y cualquier módulo que también tenga una declaración de *extern* para cualquiera de estas variables puede acceder a ellas en forma directa. El cargador asigna simplemente una localidad de almacenamiento para cada una de estas variables y guarda su dirección con cada variable de ese nombre declarada *extern*.

A los datos globales a los que se va a hacer referencia en un solo módulo se les puede aplicar el atributo *static* (estático).

Paso de parámetros. Todos los subprogramas se definen como funciones, pero los mismos pueden ser llamados ya sea como procedimientos o como funciones. Las funciones pueden devolver valores de cualquier tipo. Si una función se usa siempre como un procedimiento, se debe usar el tipo *void* (vacío) para indicar que no se devuelve un valor.

En la definición original de C, los parámetros se indicaban como:

```
p(q, r, s);
    type1 q;
    type2 r;
    type3 s;
    { ... }
```

pero el C ANSI permite:

```
q(type1 q, type2 r, type3 s)
    { ... }
```

Todos los parámetros en C son de llamada por valor. La llamada por referencia se puede simular pasando hacia adentro el valor l de un argumento usando variables apuntador. Así, en:

```
p(a, b);
...
void p(int *u, int *v) { ... }
```

a y b son llamadas por parámetros de valor, mientras en:

```
p(&a,&b);
...
void p(int *u,int*v) {*u=*v+1;...}
```

Los valores de *a* y *b* se pasan hacia adentro, y el enunciado $*u = *v + 1$ tiene el efecto de cambiar el valor de *a* de modo que sea *b* + 1.

Los parámetros de arreglo se pasan hacia adentro como *name* (nombre), puesto que no se necesita información sobre límites para tener acceso al arreglo. Los arreglos declarados con más de una dimensión (por ejemplo, *a*[10][20]), se pasan como el argumento *a*, pero tienen límites de arreglo dados en la lista de parámetros.

Funciones normales

Gran parte del poder de C es consecuencia de la rica biblioteca de funciones que ayuda a generar programas. Muchas de ellas son interfaces directas al sistema operativo. En este libro no se intenta cubrir todas esas funciones; sin embargo, las siguientes son algunas de las más importantes que el lector habrá de necesitar en programas sencillos.

Entrada y salida normales. Estas funciones están definidas en stdio.h. Para rutinas sencillas de entrada y salida, las más importantes son *getchar* para entradas de caracteres y *printf* para salidas simples con formato.

Los archivos *stdin* y *stdout* están predefinidos como el archivo normal de entrada (por lo general teclado) y el archivo normal de salida (por lo general pantalla de visualización), respectivamente. Estos nombres se pueden omitir por lo general de las listas de parámetros. Por ejemplo, la lectura de caracteres de un archivo requiere la función *getc(nombredearchivo)*, en tanto que la lectura desde el teclado requiere sólo *getchar*, sin nombre de archivo. *getchar* se define simplemente como *getc(stdin)*.

EOF se define como una constante y se usa para indicar condiciones de final de archivo (*end of file*). (Por lo común es el valor –1 para distinguirlo de cualquier carácter válido que se pueda leer.)

int getchar() devuelve el carácter siguiente del flujo de entrada (*stdin*).

int getc(FILE nombredearchivo) devuelve el carácter siguiente de *nombredearchivo*.

*FILE * fopen(char * cadenadearchivo, char * tipodearchivo)* abre el archivo *cadenadearchivo* para entradas (si *cadenadearchivo* ="*r*"), para salida nueva (si *cadenadearchivo* ="*w*"), o para anexar un archivo ya existente (si *cadenadearchivo* ="*a*"). Los archivos *stdin* (entrada normal), *stdout* (salida normal) y *stderr* (salida de error) se abren en forma automática.

putchar(char x) imprime el carácter *x* en *stdout*.

*putc(char x,FILE * filename)* hace salir *x* hacia el archivo *filename*.

*fgets(char * s, int n, FILE * nombredearchivo)* lee e introduce un arreglo de caracteres desde *nombredearchivo*. *s* es un apuntador a un arreglo de caracteres. La función lee e introduce caracteres hasta que:

1. Se ve un nuevo renglón, o
2. Se alcanza un final de archivo, o
3. Se han leído e introducido $n - 1$ caracteres

Se anexa una terminación nula \0 a la cadena en *s*.

*feof(FILE * nombredearchivo)* devuelve *TRUE* si se alcanzó un final de archivo en una lectura previa.

int printf(cadena,argumentos) imprime *cadena* en *stdout*. Para que el renglón aparezca en la terminal, *cadena* debe terminar con un carácter de final de renglón \n. Si la *cadena* contiene %d o %i, se imprime el próximo argumento de la lista de parámetros en un formato de entero. Si se trata de %s, entonces se supone que el argumento siguiente es una cadena con terminación

nula e imprime. %c significa un carácter para el argumento siguiente, %f son datos de punto flotante sin exponente explícito, mientras que %e significa imprimir el exponente *E*. %o es salida octal.

*int sprintf(char * s, cadena, argumentos)* es lo mismo que *printf* excepto que la cadena *s* se escribe dentro de *s*.

Funciones de asignación de memoria. Estas se encuentran definidas en malloc.h.

*void * malloc(int valor)* asigna un bloque de almacenamiento de tamaño *valor* y devuelve un apuntador a él.

int sizeof(nombredetipo) devuelve el tamaño del objeto de tipo *nombredetipo*. Ésta se suele emplear con *malloc*, como en:

aptvar = (tiponuevo)malloc(sizeof(tiponuevo));*

*int free(char * mallocapt)* devuelve el almacenamiento al que apunta *mallocapt*, el cual fue asignado originalmente por *malloc*.

Funciones de cadena. Éstas se encuentran definidas en string.h.

*char * strcat(char * s1, char * s2)* anexa *s2* en el extremo de *s1* y devuelve el apuntador a *s1*.

*char * strncat(char * s1, char * s2, int n)* anexa hasta *n* caracteres de *s2* al extremo de *s1* y devuelve el apuntador a *s1* (o todo *s2* si es inferior al tamaño de *n*).

*int strcmp(char * s1, char * s2)* devuelve un valor menor de 0 si *s1* es lexicográficamente menor que *s2*, 0 si son iguales, y un valor mayor de 0 si *s1* es mayor que *s2*.

int strncmp(s1, s2, n) es como *strcmp* excepto que compara sólo hasta *n* caracteres de *s2*.

*char * strcpy(char * s1, char * s2)* sobreescribe *s1* con *s2*. Se devuelve un apuntador a *s1*.

*char * strncpy(char * s1, char * s2, int n)* copia los primeros *n* caracteres de *s2* en *s1*.

*int strlen(char * cadena)* devuelve la longitud de la cadena.

Funciones de conversión. Éstas se encuentran definidas en stdlib.h.

*double strtod(char * string, char *,apt)* convierte una cadena a *float* y fija *apt* de modo que apunte al carácter que sigue al último carácter convertido de *cadena*. Si *apt* es nulo, nada se fija.

*long strtol(char * string, char **apt,int base)* convierte una cadena a un entero largo, *base* es el número base del dato de entrada. Los valores de 2, 8 y 10 son los más útiles.

Las funciones *atoi*(*char * cadena*) (ASCII a entero), *atol* (ASCII a largo) y *atof* (ASCII a 'float') tienen su origen en versiones más antiguas de C.

10.2.7 Abstracción y encapsulamiento

El lenguaje C contiene un mecanismo bastante completo para definición de tipos, pero en realidad no implementa el ocultamiento de información. Cualquier procedimiento que tiene acceso a un **struct** también lo tiene a los datos internos que constituyen el nuevo tipo.

La convención de módulos de poner definiciones de interfaz en archivos ".h" funciona bastante bien en la práctica, pero es sólo una convención del lenguaje y no un requisito.

10.2.8 Evaluación del lenguaje

El C se ha vuelto un lenguaje muy popular. Tiene una sintaxis vigorosa y clara, en tanto que la estrecha asociación de los objetos de datos en C y la arquitectura de máquina subyacente permiten escribir programas muy eficientes. La gran biblioteca de funciones normales proporciona a C una forma fácil de administrar los recursos del sistema. La relación entre arreglos y apuntadores, al igual que la visión transparente de las cadenas como arreglos de caracteres o apuntadores a series de caracteres, da cabida a operaciones eficientes de procesamiento de cadenas.

Sin embargo, C permite una programación muy descuidada y propensa a errores. Aunque tiene tipos fuertes, puesto que los enteros, caracteres, datos lógicos y tipos enumerados son todos subtipos de enteros, la tipificación fuerte no es tan eficaz. Los enunciados **break** y **switch** son particularmente propensos a errores. Una opción **switch** fluye a la siguiente si no hay un **break**. Además, puede resultar confuso de cuál construcción va a salir un **break**. De hecho, un enunciado **break** mal ubicado fue la causa de un colapso importante de la red telefónica en Estados Unidos a principios de la década de 1990. El gran número de operaciones permite escribir operaciones similares en numerosas formas que ocasionan confusión.

El C no tiene un verdadero concepto de encapsulamiento, aunque cuenta con tipos estructurados. El C++ incorpora esas características al lenguaje.

Para el novato, el uso erróneo del operador de asignación = en vez del operador de igualdad == y el uso equivocado de paso de parámetros de llamada por valor cuando se necesitan apuntadores para implementar llamada por referencia es una fuente constante de exasperación al aprender el lenguaje.

¿Por qué se ha vuelto el C tan popular? Como señala Stroustrup, diseñador del C++ [STROUSTRUP 1986]:

"Es claro que C no es el lenguaje más limpio que se ha desarrollado ni el más fácil de usar, así que, ¿por qué lo usa tanta gente?

1. Es flexible [su uso en cualquier área de aplicación] ...
2. Es eficiente [debido a la semántica de bajo nivel del lenguaje] ...
3. Está disponible [debido a la ubicuidad del compilador de C esencialmente en toda plataforma de cómputo] ...

Capítulo 13

Lenguajes funcionales

Los lenguajes de este capítulo, LISP y ML, difieren de los anteriores en dos aspectos importantes. Desde el punto de vista de su diseño, están destinados a programarse en forma aplicativa. La ejecución de una expresión es el concepto principal que se necesita para la secuenciación de programas, en vez de los enunciados de lenguajes como C, Pascal o Ada. Puesto que son aplicativos (o funcionales), el almacenamiento en montículos y las estructuras de lista se convierten en el proceso natural para gestión de almacenamiento en lugar del mecanismo tradicional de registros de activación de los lenguajes de procedimiento.

LISP representa uno de los primeros lenguajes de programación. Fue diseñado alrededor de 1960, en la misma época que el FORTRAN original. Sus objetos básicos son átomos y listas de átomos, y la llamada y recursión de funciones constituyen los mecanismos básicos de ejecución, como ya se ha descrito en el capítulo 6. El lenguaje ML, por otra parte, emplea una estructura de control, que tiene el aspecto del C o del Pascal, para operar sobre datos numéricos y listas.

La diferencia principal entre LISP y ML es que el concepto de *tipo* es fundamental para el diseño del ML. Éste tiene un concepto avanzado de polimorfismo, el cual amplía considerablemente los conceptos **contains** del FORTRAN 90, **generic** del Ada y **template** del C++.

13.1 LISP

LISP fue proyectado e implementado inicialmente por John McCarthy y un grupo del Massachusetts Institute of Technology alrededor de 1960 [MCCARTHY 1961]. El lenguaje se ha llegado a usar ampliamente para investigación en ciencias de la computación, en forma destacada en el área de inteligencia artificial (IA: robótica, procesamiento de lenguajes naturales, prueba de teoremas, sistemas inteligentes, etc.). Se han desarrollado muchas versiones de LISP a lo largo de los últimos 30 años y, de todos los lenguajes que se describen en este libro, LISP es el único que no está estandarizado ni dominado por una implementación particular.

LISP es diferente de casi todos los demás lenguajes en varios aspectos. El más notable es la equivalencia de forma entre programas y datos en el lenguaje, la cual permite ejecutar estructuras de datos como programas y modificar programas como datos. Otra característica destacada es la fuerte dependencia de la recursión como estructura de control, en vez de la iteración (formación de ciclos) que es común en casi todos los lenguajes de programación.

13.1.1 Historia

Como ya se ha señalado, LISP tuvo sus inicios en el MIT alrededor de 1960. Durante los años sesenta y setenta se desarrollaron varias versiones del lenguaje. Éstas incluian el MacLisp del MIT, el Interlisp de Warren Teitelman para la DEC PDP-10, el Spice LISP y el Franz LISP. El Interlisp fue el dialecto dominante durante la mayor parte de esta época. Durante la parte final de la década de 1970, Gerald Sussman y Guy Steele desarrollaron una variante, como parte de una investigación sobre modelos de computación, que se conocía como "Schemer", pero, debido a limitaciones a un nombre de seis caracteres, se abrevió a simplemente Scheme. Ésta es la versión que ha tenido el máximo impacto en el uso universitario del lenguaje.

En abril de 1981, se llevó a cabo una reunión entre las diversas facciones del LISP para tratar de fusionar los diversos dialectos en un solo lenguaje LISP, el cual se difundió con el nombre de Common LISP, a falta de un mejor nombre "común". El nombre "Standard LISP" ya había sido adoptado por uno de estos dialectos. La estructura básica del Common LISP se desarrolló a lo largo de los tres años siguientes.

La época desde 1985 hasta principios de los años noventa representó probablemente la de máxima popularidad para LISP. La investigación en IA prosperaba, y en muchas universidades, quizá el 75% o más de los estudiantes de posgrado declaraban la IA como su área de especialización. En 1986, el grupo técnico de trabajo X3J13 se reunió para estandarizar el Common LISP, y se inició un esfuerzo por fusionar los dialectos Scheme y Common LISP. Esto fracasó, y en 1989 el IEEE desarrolló un estándar propio para Scheme. Alrededor de esta época, los efectos de la orientación a objetos en lenguajes como C++ y Smalltalk se estaban haciendo sentir, y se escribió el Common LISP Object System (CLOS; Sistema de Objetos en Common LISP). Finalmente, en 1992, el X3J13 desarrolló un borrador para Common LISP de más de 1000 páginas, más grande que el estándar para COBOL. Parece extraño que un lenguaje con un diseño básico tan sencillo y limpio pudiera crecer fuera de control.

Desde un principio, LISP sufrió críticas por ejecución lenta, en especial en la computadora estándar de von Neumann descrita en el capítulo 2. Así como las funciones originales *car* y *cdr* tuvieron como modelo el hardware de la IBM 704, se estaban ideando arquitecturas de máquina alternativas para acelerar la ejecución del LISP. Varias compañías desarrollaron máquinas proyectadas para la ejecución rápida de LISP. Sin embargo, alrededor de 1989 se desarrolló una estrategia de recolección de basura para arquitecturas normales de von Neumann que era competitiva con el hardware de LISP para usos especiales. Por esta razón, ninguna de las compañías de LISP sobrevivió hasta llegar a ser un éxito comercial a largo plazo.

13.1.2 Hola Mundo

En este libro se hará énfasis en el dialecto Scheme de LISP, aunque se señalarán otros dialectos, como el Common LISP. LISP suministra un modelo sencillo de ejecución de programas que difiere de casi todos los demás lenguajes que comúnmente se programan. LISP se ejecuta por lo general bajo el control de un intérprete. La figura 13.1 proporciona nuestro programa trivial "Hola Mundo".

```
%lisp
>(define (hola i) (cond ((eq i 2) (print "Hola Mundo"))
            (T (print "Adiós Mundo"))))
 HOLA
>(define (hazlo) (hola 2))
HAZLO
>(hazlo)
"Hola Mundo"
"Hola Mundo"
```

Figura 13.1. "Hola Mundo" en LISP.

El símbolo inicial > es la señal de entrada de LISP, y el renglón que sigue a cada entrada es el valor resultante de esa expresión, por ejemplo, el valor de la función `print` es la cadena que se imprime.

13.1.3 Breve perspectiva del lenguaje

Los programas en LISP se ejecutan en un ambiente interactivo (ordinariamente) y, como resultado, no existe un programa principal en la forma usual. En su lugar, el usuario que está en una terminal introduce el "programa principal" como una sucesión de expresiones por evaluar. El sistema LISP evalúa cada expresión conforme se introduce e imprime el resultado en forma automática en la terminal. Ordinariamente, algunas de las expresiones introducidas son definiciones de función. Otras expresiones contienen llamadas a estas funciones definidas con argumentos particulares. No existe una estructura de bloques u otra organización sintáctica compleja. Las únicas interacciones entre diferentes funciones ocurren a través de llamadas durante la ejecución.

Las funciones en LISP se definen enteramente como expresiones. Cada operador es una función que devuelve un valor, y los subprogramas se escriben como expresiones únicas (que suelen ser muy complejas). Se han incorporado diversas construcciones especiales al lenguaje para hacer que esta sintaxis de expresión pura se parezca en cierto modo a la sintaxis ordinaria de *serie de enunciados,* pero la forma de expresión sigue siendo básica.

Los datos en LISP son más bien restringidos. Los *átomos literales* (símbolos) y los *átomos numéricos* son los tipos elementales básicos. Las listas vinculadas y listas de propiedades (representadas como un caso especial de listas vinculadas) constituyen las estructuras de datos básicas. Todo el procesamiento de descriptores se hace durante la ejecución y no se necesitan declaraciones de ningún tipo.

El LISP suministra una amplia variedad de primitivas para la creación, destrucción y modificación de listas (incluso listas de propiedades). Se proveen primitivas básicas para aritmética. La traducción y ejecución de programas en tiempo de ejecución también se suministran como primitivas, y los programas se pueden crear y ejecutar en forma dinámica.

Las estructuras de control en LISP son relativamente sencillas. Las expresiones que se usan para construir programas se escriben en forma polaca estricta de Cambridge y pueden incluir

bifurcación condicional. La característica *prog* suministra una estructura sencilla para escribir expresiones en una secuencia. Se hace fuerte énfasis en las llamadas de función recursivas en casi toda la programación en LISP.

Los comentarios en LISP comienzan ordinariamente con un punto y coma, que indica que el resto del renglón es un comentario.

Las referencias en LISP se basan primordialmente en la regla de la *asociación más reciente* para referencias no locales, la cual se suele implementar usando una lista vinculada simple de asociaciones vigentes, la *lista A*, en la cual se busca la asociación vigente cada vez que se hace referencia a un identificador. Con todo, se proveen algunos giros para hacer posible el reemplazo de esta técnica sencilla aunque lenta por métodos más rápidos. El más importante permite dar a cualquier identificador una asociación global, que tiene prioridad sobre cualquier otra asociación del identificador.

Los parámetros de función se transmiten ya sea todos por valor o todos por nombre, según la clasificación de la función; la transmisión por valor es el caso usual.

LISP se implementa más fácilmente con un intérprete de software y simulación de software para todas las primitivas. Casi todas las implementaciones también suministran un compilador que se puede usar para compilar definiciones de función seleccionadas a código de máquina. Estas funciones compiladas son entonces ejecutables por el intérprete de hardware (pero de cualquier forma requieren simulación de software para muchas operaciones). LISP es poco idóneo para compilación porque la mayoría de los enlaces no se forman sino hasta la ejecución. Se usa una estructura compleja de gestión de almacenamiento con base en un montículo con recolección de basura como almacenamiento primario para datos y programas.

Ejemplo anotado

Una vez más, nuestro ejemplo anotado de la figura 13.2 suma un arreglo de enteros. En este caso, se digitan las funciones directamente en LISP. Nuestro LISP particular tiene la función (*load "nombredearchivo"*), la cual lee las definiciones de funciones de LISP desde un archivo. El símbolo > es la señal de entrada de LISP.

Renglón 1. Invocar el intérprete de LISP.

Renglón 2. Un comentario en LISP va desde el; hasta el final del renglón.

Renglón 3. `SumarSiguiente` es una función de la cual se declara que tiene un parámetro `V`. Devuelve la suma de las entradas de su parámetro. La definición para `SumarSiguiente` es la expresión de los renglones 4-5.

Renglón 4. `cond` busca la primera pareja <*predicado, expresión*> que es cierta. Esta primera pareja tiene el predicado `null V`, que será cierto después que se haya procesado el último valor de entrada. La función imprime `SUMA=` y devuelve.

La construcción `progn` ejecuta en orden una serie de expresiones, en este caso la función `print` seguida de la expresión 0 como valor por omisión.

Renglón 5. Si el vector `V` no es nulo (el caso ordinario), el predicado `t` se vuelve el caso por omisión para la `cond` del renglón 4. El vector `V` menos el primer miembro se pasa a `SumarSiguiente` en forma recursiva y, al devolver, la cabeza de `V` se suma a la suma previamente calculada.

```
1  %lisp
2  >; Guardar valores como una lista de caracteres
3  >(define (SumarSiguiente V)
4        (cond ((null V) (progn (print "Suma=") 0))
5              (T  (+ ( SumarSiguiente (cdr V)) (car V)) ) ))
6  SUMARSIGUIENTE
7  >; Crear vector de valores de entrada
8  (defun ObtenerEntrada(f c)
9              (cond ((eq c 0) nil)
10             (T (cons (read f) (ObtenerEntrada f (- c 1))))))
11 OBTENERENTRADA
12 >(defun Hazlo()
13             (progn
14             ( setq archivoent (open '"lisp.data"))
15             ( setq arreglo (ObtenerEntrada archivoent (read archivoent)))
16             (print arreglo)
17             (print (SumarSiguiente arreglo))))
18 HAZLO
19 >Hazlo
20
21 (1 2 3 4)
22 "Suma="
23 10
24 10
```

Figura 13.2. Suma de un vector de números en LISP.

Renglón 6. LISP imprime el nombre de la función `SUMARSIGUIENTE` como resultado de los renglones 3-5.

Renglones 7-11. Estos renglones definen `ObtenerEntrada`, que tiene dos parámetros, el apuntador de archivo de entrada, `f`, y la cuenta de elementos de vector que quedan por leer, `c`. Esta definición de función muestra la sintaxis `defun` para definición de funciones en vez de la `define` de Scheme. `ObtenerEntrada` devuelve una lista de los valores de datos que se introducen por lectura. Por ejemplo, un arreglo de entrada con los valores 1 2 3 4 tendrá la estructura que se muestra en la figura 13.3.

Renglón 9. La `cond` verifica primero que la cuenta `c` sea 0 y, si lo es, devuelve la lista nula.

Renglón 10. Si la cuenta no es 0, `ObtenerArchivo` crea un nuevo vector leyendo un átomo del archivo `f` y lo agrega a la cabeza de la lista devuelta de la llamada recursiva a `ObtenerEntrada` con parámetros `f` y la cuenta disminuida en 1.

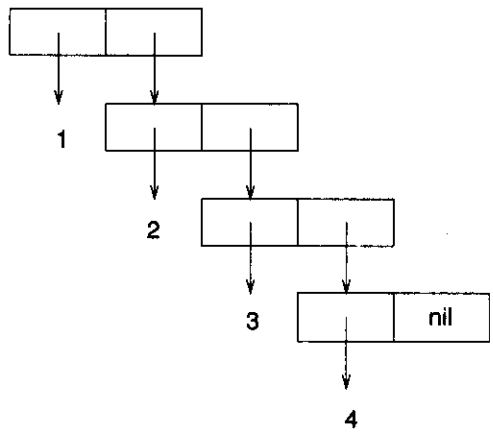


Figura 13.3. Muestra de estructura de datos de listas en LISP.

Renglones 12-18. Esto define la función principal de control para el programa. `Hazlo` no tiene parámetros.

Renglón 13. Aquí, `progn` causa que las expresiones de los renglones 14-17 se ejecuten en orden.

Renglón 14. La función `open` abre el archivo `lisp.data` y devuelve un apuntador de archivo. `setq` asigna este apuntador de archivo a la variable archivoent. La secuencia de acciones es que se evalúa el argumento open y luego se hace la asignación a `archivoent`.

Renglón 15. Se llama la función `ObtenerEntrada` (renglones 8-10) con parámetros `archivoent`, el cual se definió en el renglón 14, y el resultado de la función read, que será la cuenta del número de elementos de vector por sumar. `setq` asigna estos datos de entrada a la variable `arreglo`.

Renglón 16. La función print exhibe el valor de los datos de entrada.

Renglón 17. Los datos de entrada (lista `arreglo`) se pasan a la función `SumarSiguiente` definida en los renglones 3-5, la cual devuelve la suma de los miembros de la lista.

Renglón 19. Se llama la función `Hazlo`.

Renglones 20-24. Si `lisp.data` contiene la entrada:

```
4 1 2 3 4
```

entonces los renglones 20-23 proporcionan la salida que se produce. El número 10 se imprime dos veces en este ejemplo, una como resultado de la función `print` del renglón 17 y otra como el valor devuelto de la llamada a `Hazlo`. La función `print` devuelve el objeto impreso como su valor.

13.1.4 Objetos de datos

Tipos de datos primitivos

Los tipos primarios de objetos de datos en LISP son listas y átomos. Las definiciones de función y las listas de propiedades son tipos especiales de listas de particular importancia. También se suministran ordinariamente arreglos, números y cadenas, pero estos tipos desempeñan un papel inferior.

Variables y constantes. En LISP, un **átomo** es el tipo elemental básico de un objeto de datos. A un átomo se le suele llamar *átomo literal* para distinguirlo de un número (o *átomo numérico*), el cual también es clasificado como átomo por casi todas las funciones en LISP. Sintácticamente, un átomo es tan solo un identificador, una cadena de letras y dígitos que se inicia con una letra. No se suele tomar en cuenta la distinción de mayúsculas y minúsculas para identificadores. Dentro de las definiciones de función en LISP, los átomos tienen los usos ordinarios de los identificadores; se emplean como nombres de variables, nombres de funciones, nombres de parámetros formales, etcétera.

Sin embargo, un átomo en LISP no es simplemente un identificador en tiempo de ejecución. Un átomo es un objeto de datos complejo representado por una posición en la memoria, el cual contiene el descriptor de tipo para el átomo junto con un apuntador a una *lista de propiedades*. La lista de propiedades contiene las diversas propiedades asociadas al átomo, una de las cuales es siempre su *nombre de impresión*, que es la cadena que representa el átomo para entradas y salidas. Otras propiedades representan diversos enlaces para el átomo, las cuales pueden incluir la función nombrada por el átomo y otras propiedades asignadas por el programa durante la ejecución.

Siempre que un átomo aparece como componente de otro objeto de datos, como una lista, está representado por un apuntador en la posición de memoria que sirve como representación del átomo en tiempo de ejecución. Por tanto, toda referencia al átomo *ABC* durante la ejecución de un programa en LISP aparece como un apuntador a la misma posición de memoria.

Todo átomo también aparece ordinariamente como componente en una tabla central, definida por el sistema, llamada *lista de objetos* (*ob_list*). La *ob_list* está organizada por lo común como una tabla codificada por dispersión que permite la búsqueda eficiente de un nombre de impresión (cadena de caracteres) y la recuperación de un apuntador al átomo que tiene ese nombre de impresión. Por ejemplo, cuando se introduce una lista que contiene la cadena de caracteres "ABC", la cual representa el átomo *ABC*, la función *read* busca en la *ob_list* la entrada "ABC", la cual también contiene un apuntador a la localidad de almacenamiento para el átomo *ABC*. Este apuntador se inserta en la lista que se está construyendo, en el punto apropiado.

Se pueden usar **números** (átomos numéricos) en formato entero o de punto flotante. Se emplea la representación de hardware, pero también se requiere un descriptor en tiempo de ejecución, de modo que cada número utiliza típicamente dos palabras. Sin embargo, esta representación se coordina bien con la que se usa para átomos literales; un número es un átomo con un designador de tipo especial y un apuntador a la cadena de bits que representa el número, en vez de un apuntador a una lista de propiedades.

Las **cadenas** en LISP se representan como "cadena de símbolos". Téngase presente que la comilla simple (') se interpreta como la función *quote* (citar) para argumentos literales (no evaluados) para funciones.

Listas de propiedades. Cada átomo literal tiene una lista de propiedades asociada, accesible a través de un apuntador guardado en la posición de memoria que representa el átomo. Una lista de propiedades es simplemente una lista ordinaria en LISP que difiere sólo en que sus elementos están apareados lógicamente en una sucesión alternante de *nombre de propiedad/valor de propiedad.* La lista de propiedades de cada átomo contiene al menos el *pname* de la propiedad, cuyo valor asociado es un apuntador a una lista que contiene el nombre de impresión del átomo en forma de cadena de caracteres. Si un átomo es un nombre de función, su lista de propiedades contiene el nombre de propiedad que proporciona el tipo de función y un apuntador a la lista que representa la definición de la función. El programador puede agregar otras propiedades según lo desee, y ciertas primitivas también agregan propiedades. En gran parte de la programación en LISP, las listas de propiedades de átomos son las estructuras centrales donde se guarda gran parte de los datos.

Tipos de datos estructurados

Las listas en LISP son estructuras simples con un sólo vínculo, como se muestra en la figura 13.3. Cada elemento de lista contiene un apuntador a un elemento de datos y un apuntador al elemento de lista siguiente. El último elemento de lista señala al átomo especial *nil* (nada) como su sucesor. Los dos apuntadores de una lista de elementos se conocen como el apuntador *car* y el apuntador *cdr*. El apuntador *cdr* señala al sucesor del elemento de lista. El apuntador *car* señala al elemento de datos.

Un elemento de lista puede contener (tener como apuntador *car*) un apuntador a un objeto de datos de cualquier tipo, incluso un átomo literal o numérico, o un apuntador a otra lista. Cada caso se distingue por un indicador de tipo de datos guardado en la localidad a la que se señala. Puesto que un elemento de lista puede contener un apuntador a otra lista, es posible construir estructuras de listas de complejidad arbitraria, como ya se ha descrito en la sección 4.3.7.

Casi todas las implementaciones de LISP suministran algún tipo de objeto de datos vector o arreglo. Sin embargo, es poca la uniformidad de tratamiento entre distintas implementaciones, lo que refleja la relativa falta de importancia de los vectores y arreglos en la mayor parte de la programación en LISP. Una implementación típica suministra una función *mkvect(límite)* que crea un objeto de datos vector con un ámbito de subíndices de 0 hasta un *límite* dado. Cada componente del vector puede contener un apuntador a cualquier objeto de datos en LISP; inicialmente, todos los apuntadores se fijan en *nil* (nada). Una función *getv(vector, subíndice)* devuelve el apuntador guardado en el componente indicado del vector argumento; por tanto, *getv* es la versión en LISP de la subindización para recuperar el valor de un componente de vector. *putv* se usa para asignar un nuevo valor a un componente de vector.

Las funciones definidas por el programador se escriben en forma de listas para entrada, como se puede ver en la figura 13.2. Las rutinas de entrada del LISP no hacen distinción entre definiciones de función y listas de datos, sino que simplemente traducen todas las listas a la representación interna de listas vinculadas. Cada átomo que se encuentra se busca en la lista *ob_list* y se recupera el apuntador al átomo si ya existe, o se crea un átomo nuevo en caso contrario.

Inicialización y asignación. La asignación directa no desempeña un papel tan fundamental en la programación en LISP como en otros lenguajes. Muchos programas en LISP se escriben por completo sin operaciones de asignación, empleando recursión y transmisión de parámetros para obtener el mismo efecto en forma indirecta. Sin embargo, se usa asignación dentro de segmentos *prog*, donde

el programa en LISP adopta la forma ordinaria de sucesión de enunciados. El operador básico de asignación es *setq*. La expresión (*setq x val*) asigna el valor de *val* como el nuevo valor de la variable *x*; el resultado de la expresión es el valor de *val* (por tanto, *setq* es una función, pero su valor se pasa ordinariamente por alto). El *set* (conjunto) primitivo es idéntico a *setq* excepto que la variable (que es simplemente un átomo) a la cual se hace la asignación se puede computar. Por ejemplo, (*set* (*car L*) *val*) es equivalente a (*setq x val*) si el átomo *x* resulta ser el primer elemento de la lista *L*. *rplaca* y *rplacd* permiten la asignación de cualquier elemento de lista a los campos *car* y *cdr*, respectivamente. Por ejemplo, (*rplaca L val*) asigna el valor de *val* como nuevo primer elemento de la lista *L*, reemplazando el primer elemento vigente.

Representación de almacenamiento

Dos características son comunes a todos los objetos de datos en LISP:

1. Cada objeto de datos tiene un descriptor en tiempo de ejecución que proporciona su tipo y otros atributos.
2. Si un objeto de datos tiene componentes (es decir, es un objeto de datos estructurado), entonces los mismos casi nunca se representan de manera directa como parte del objeto de datos; en su lugar, se usa un apuntador al objeto de datos componente.

13.1.5 Control de secuencia

LISP no distingue entre datos en LISP y programas en LISP, lo que conduce a un diseño sencillo para el intérprete de LISP. La ejecución de LISP es amena, comparada con la de otros lenguajes de este libro, en cuanto a que el intérprete básico se puede describir en LISP. A la función *apply* (aplicar) se le pasa un apuntador a la función por ejecutar, y *apply* interpreta la definición de esa función, empleando una segunda función *eval* para evaluar cada argumento.

El traductor de LISP es simplemente la función *read* (leer). *read* examina una cadena de caracteres proveniente del archivo de entrada o terminal, en busca de un identificador o estructura de lista completa. Si se encuentra un solo identificador, se busca en la *ob_list* para obtener un apuntador al átomo correspondiente (o para crear un nuevo átomo si no se encuentra uno con ese nombre de impresión). Si se encuentra una estructura de lista que comience con "(", entonces se examina cada elemento de la lista hasta que se encuentra un ")" complementario. Cada elemento de lista se traduce a forma interna y se inserta un apuntador en la lista como apuntador *car* al punto apropiado. Los números se traducen a sus equivalentes binarios, con un descriptor en una palabra por separado. Si se encuentra una sublista que comience con otro "(", entonces se llama *read* de manera recursiva para construir la representación interna de la sublista. En seguida, se inserta un apuntador a la sublista como apuntador *car* del componente siguiente de la lista principal. Adviértase que este proceso de traducción no depende de si la lista es una definición de función o una lista de datos; todas las listas se tratan en la misma forma.

La ejecución de un programa en LISP consiste en la evaluación de una función en LISP. La secuenciación tiene lugar normalmente por llamada de función (que puede ser recursiva) y por expresión condicional.

Expresiones

Un programa en LISP se compone de una sucesión de definiciones de función, donde cada función es una expresión en notación polaca de Cambridge.

Condicionales. La condicional es la estructura principal para proporcionar una secuenciación alternativa de ejecución en un programa en LISP. La sintaxis es:

(cond *alternativa*$_1$
alternativa$_2$
...
alternativa$_n$
(T *expresión_por_omisión*))

donde cada *alternativa*$_i$ es (*predicado*$_i$ *expresión*$_i$).

cond se ejecuta evaluando cada *predicado*$_i$, por turno, y evaluando la *expresión*$_i$ del primero que devuelva cierto (*T*). Si todos los predicados son falsos, se evalúa *expresión_por_omisión*.

Operaciones sobre átomos. Las operaciones sobre átomos incluyen típicamente la función de prueba *atom*, que distingue entre un apuntador a una palabra de lista y un apuntador a un átomo (verificando el descriptor de la palabra); *numberp*, que prueba si un átomo es un átomo literal o un número; *eq*, que prueba si dos átomos literales son iguales (probando simplemente si sus dos argumentos señalan a la misma posición); *gensym*, que genera un átomo nuevo (y no lo pone en la *ob_list*); *intern*, que pone un átomo en la *ob_list*; y *remob*, que elimina un átomo de la *ob_list*.

LISP contiene las primitivas aritméticas básicas: +, –, *, / y unas cuantas más. La sintaxis es la misma que para otras operaciones en LISP; así, $A + B * C$ se escribe (+ A (* *B C*)). Todas las operaciones aritméticas son operaciones genéricas que aceptan argumentos de tipo de datos ya sea real o entero y hacen las conversiones de tipo según se requiere.

La operación relacional *zerop* prueba en busca de un valor de cero; *greaterp* y *lessp* representan las comparaciones usuales mayor que y menor que. Los resultados de estas operaciones son ya sea el átomo *nil*, que representa *false* (falso), o *T*, que representa *true* (cierto) (cualquier valor que no sea *nil* representa ordinariamente *true* en LISP).

No existe un tipo de datos booleano en LISP. Las operaciones booleanas *and*, *or* y *not* se suministran como funciones. La operación *and* toma una lista arbitraria de argumentos no evaluados; evalúa cada uno por turno y devuelve un resultado *nil* si el resultado de la evaluación de cualquiera de sus argumentos es *nil*. La operación *or* funciona de manera similar.

Operaciones sobre listas. Las primitivas *car* y *cdr* recuperan el apuntador *car* y el apuntador *cdr* de un elemento de lista dado, respectivamente. En efecto, dada una lista *L* como operando, (*car L*) devuelve un apuntador al primer elemento de la lista, y (*cdr L*) devuelve una apuntador a la lista con el primer elemento eliminado.

La primitiva *cons* toma dos apuntadores como operandos, asigna una nueva palabra de memoria de elemento de lista, guarda los dos apuntadores en los campos *car* y *cdr* de la palabra y devuelve un apuntador a la nueva palabra. Cuando el segundo operando es una lista el efecto consiste en agregar el primer elemento a la cabeza de la lista y devolver un apuntador a la lista ampliada.

car, *cdr* y *cons* son las operaciones básicas para seleccionar componentes de lista y construir listas. A través del uso de *cons*, se puede construir cualquier lista elemento por elemento. Por ejemplo:

(cons A (cons B (cons C nil)))

construye una lista de los tres elementos a los que se hace referencia con *A*, *B* y *C*. En forma similar, si *L* = (*A B C*) es una lista, entonces (*car L*) es *A*, (*car* (*cdr L*) es *B* y (*car* (*cdr* (*cdr L*))) es *C*. Si se usan *car*, *cdr* y *cons* de manera apropiada, cualquier lista se puede descomponer en sus elementos constitutivos, y se pueden construir nuevas listas a partir de éstos u otros elementos.

La primitiva *list* se puede usar para reemplazar una serie larga de operaciones *cons*. *list* toma cualquier número de argumentos y construye una lista de argumentos, para luego devolver un apuntador a la lista resultante.

quote permite escribir cualquier lista o átomo como una literal en un programa, como se describe en la sección 4.3.7.

La primitiva *replace* se usa para reemplazar el campo de apuntador *car* de una palabra de lista por un nuevo apuntador; *replacd* se usa para cambiar el apuntador *cdr*. Estas dos primitivas se deben usar con cuidado en LISP porque modifican en efecto el contenido de una palabra de lista además de devolver un valor y, por tanto, tienen *efectos colaterales*. A causa de la complejidad de la manera como se vinculan las listas en LISP, estos efectos colaterales pueden afectar a otras listas además de la que se está modificando.

null prueba si una lista está vacía (si es igual al átomo *nil*), *append* se puede usar para concatenar dos listas y *equal* compara dos listas en cuanto a igualdad de elementos correspondientes (aplicándose ella misma en forma recursiva a parejas de elementos correspondientes).

Primeras propiedades en operaciones. Se suministran funciones básicas para inserción, eliminación y acceso a propiedades de listas de propiedades. *put* se usa para agregar una pareja de nombre/valor de propiedad a una lista de propiedades; *get* devuelve el valor vigente asociado con un nombre de propiedad dado, y *remprop* elimina una pareja de nombre/valor de una lista de propiedades. Por ejemplo, para agregar la pareja de nombre/valor (*edad*, 40) a la lista de propiedades del átomo *maría*, uno escribe:

(*put* '*maría* '*edad* 40)

Más adelante en el programa, la propiedad *edad* de *maría* se puede recuperar a través de la llamada de función:

(*get* '*maría* '*edad*)

La eliminación de la propiedad *edad* de *maría* se consigue a través de la llamada:

(*remprop 'maría 'edad*)

Las primitivas que definen funciones también pueden modificar listas de propiedades para nombres de propiedad especiales, tales como *expr* y *fexpr* (véase más adelante).

Enunciados

LISP carece de un concepto real de ejecución de enunciados. Este lenguaje se ejecuta evaluando funciones. El grupo de llamadas de función normales representa el conjunto usual de "estructuras de control". La expresión *cond* es prácticamente el único mecanismo de secuenciación de ejecución de primitivas que se necesita.

La función *prog* proporciona los medios para la ejecución secuencial, pero no es absolutamente necesaria, aunque facilita la escritura de casi cualquier programa en LISP. Una *prog* tiene la sintaxis:

(*prog*(*variables*)
(*expresión*$_1$)
(*expresión*$_2$)
...
(*expresión*$_n$))

donde *variables* es una lista de variables que es conocida dentro de la sucesión de expresiones, y cada *expresión*$_i$ se ejecuta en orden.

(*progn*(*expresión*$_1$)...) es similar, sin las variables locales.

Se puede salir de una *prog* ya sea completando la evaluación de la última expresión (en cuyo caso la *prog* tiene en conjunto el valor *nil*) o por una llamada a la primitiva *return*. El argumento para *return* es el valor que debe ser devuelto como valor de la *prog*.

Entrada y salida

Las entradas y salidas son simplemente llamadas de función. (*read*) lee el átomo siguiente proveniente del teclado. Los renglones 14 y 15 de la figura 13.2 muestran cómo un archivo de datos puede ser abierto por la función *open* y leído a través de la llamada (*read apuntador_de_archivo*).

(*print objeto*) imprime un objeto en un formato legible.

Definición de funciones

La función *defun* se usa para crear nuevas funciones. La sintaxis de esta función es:

(*defun nombre_de_función*(*argumentos*)*expresión*)

donde *argumentos* son los parámetros para la función, y el cuerpo de la misma es una sola expresión (que puede ser una sucesión de *progn* u otras secuencias de ejecución complejas).

Alternativamente, en Scheme, la sintaxis:

(*define* (*nombre_de_función argumentos*)*expresión*)

se puede usar para definir una función.

Cuando LISP inició su desarrollo a principios de los años sesenta, la definición de función se basaba en la expresión lambda, como se expuso en la sección 9.4.1. Se pueden usar expresiones lambda para definir funciones:

(*define nombre_de_función* (*lambda*(*parámetros*)(*cuerpo*)))

donde el nombre de función y cada nombre de parámetro son simplemente átomos, y el cuerpo es cualquier expresión donde intervienen funciones primitivas o definidas por el programador.

La acción de la declaración de funciones es bastante simple. Las definiciones de funciones interpretadas se guardan ordinariamente como una pareja de atributo-valor en la lista de propiedades del átomo que representa el nombre de la función. Para una función ordinaria, como la que se acaba de definir, se utiliza el nombre de atributo *expr*, y el valor asociado es un apuntador a la estructura de listas que representa la definición de la función. Por tanto, la llamada anterior a *define* es equivalente a la llamada:

(*put* '*nombre_de_función* '*expr* '(*lambda*(*parámetros*)(*cuerpo*)))

Muchos sistemas en LISP reservan una localidad especial en el átomo (bloque de cabecera) para la definición de la función que se nombra con ese átomo, para evitar la necesidad de buscar la definición de la función en la lista de propiedades cada vez que se llama la función. La localidad especial designa el tipo de función (*expr*, *fexpr*, etc.) y contiene un apuntador a la estructura de listas que representa su definición de función. En estas implementaciones, por lo común se suministran primitivas especiales para obtener, insertar y eliminar directamente una definición de función de la localidad especial.

13.1.6 Gestión de subprogramas y almacenamiento

LISP no suministra mecanismos de abstracción más allá de los subprogramas de función. No cuenta con estipulaciones para tipos definidos por el programador o abstracciones de datos. Casi todas las implementaciones de LISP distinguen tres clases de funciones: *funciones interpretadas*, *funciones compiladas* y *macros*.

Una **función interpretada** es aquella cuya definición se representa en forma de estructura de listas. El intérprete de software, representado por las primitivas *eval* y *apply*, se usa para ejecutar la definición de función cuando se llama la función. Son posibles dos modos de transmisión de parámetros en cada función, ya sea evaluando todos los parámetros reales antes de la transmisión o no evaluando ninguno de ellos. La función se clasifica como una función *expr* si sus argumentos siempre se evalúan, y una función *fexpr* si sus argumentos nunca se evalúan. Casi todas las funciones definidas por el programador son *expr*.

Cuando una función interpretada se llama con una lista particular de parámetros reales, el sistema de LISP invoca *apply*. *apply* evalúa los argumentos y los agrega al ambiente de referencia vigente. *apply* llama entonces a *eval* para evaluar efectivamente la expresión que constituye el cuerpo de la función. *eval* recorre la estructura de listas que representa el cuerpo de la función; evalúa los átomos que representan variables buscando su valor vigente en el ambiente de referencia y evalúa las llamadas internas de función llamando *apply* (recursivamente) con el nombre de la función y la lista de argumentos. Así pues, *eval* y *apply* interactúan una con otra para recorrer la estructura de listas que representa la definición de función y determinar en último término el valor de la expresión que representa la estructura de listas.

Las **funciones compiladas** son aquellas que han sido compiladas a un bloque de código de máquina que puede ser ejecutado por el intérprete de hardware. La representación en tiempo de ejecución de una definición de función compilada tiene una pareja de atributo-valor guardada en la lista de propiedades del átomo que representa el nombre de la función (o en una localidad especial en la cabeza del átomo), de igual manera que para una función interpretada. Sin embargo, el nombre del atributo es *cexpr* (*expr* compilada) o *cfexpr* (*fexpr* compilada), y el valor asociado es un apuntador a un bloque especial de almacenamiento que contiene el código compilado. *eval* y *apply*, cuando se les da una función compilada para ejecutarla, invocan el intérprete de hardware para ejecutar el código compilado una vez que han transmitido los argumentos en forma apropiada para una función compilada. El cuerpo de la función compilada puede contener llamadas a funciones interpretadas, en cuyo caso *eval* y *apply* son invocadas de manera recursiva por llamadas provenientes del intérprete de hardware.

Se usan diversos métodos para compilar una definición de función en las distintas versiones de LISP. Se puede suministrar una primitiva especial *compile* o, más comúnmente, se puede usar una variable especial definida por el sistema, por ejemplo, *comp*. El programador fija *comp* en *nil* o no *nil*. Cuando se llama a *define*, si *comp* es *nil* entonces *define* define la función como función interpretada: si es no *nil*, la definición se compila.

La tercera clase general de función en LISP es la *macro*. Una vez más, se usa la primitiva *define* para definir esta clase de función, pero *macro* debe tomar el lugar de *lambda* en la definición de función. Una macro es simplemente una función ordinaria en LISP (puede ser interpretada o compilada), pero su ejecución tiene un giro especial. Una macro tiene siempre un solo argumento, el cual consiste en la expresión completa que invocó la macro, incluso el nombre de la macro y la lista de parámetros reales sin evaluar. En vez de evaluar esta expresión, la función macro construye una segunda expresión equivalente como su resultado y el sistema de LISP evalúa entonces esta expresión resultante en lugar de la expresión original. El proceso de sustitución de la expresión de llamada de macro original por la expresión resultante se conoce como *expansión de macro*. Así, cuando el intérprete de LISP encuentra una llamada de macro, se invoca una secuencia de evaluación en dos pasos: primero se llama la función macro y luego se evalúa el resultado de esa llamada para obtener el resultado final.

Por ejemplo, un uso común de las macros es el encaminado a suministrar una sintaxis más legible para las operaciones usuales en LISP. Una sintaxis **if...then...else** para la condicional usual *cond* podría emplear la sintaxis:

$$(IF\ (prueba)\ THEN\ (expr_1)\ ELSE\ (expr_2))$$

Adviértase que ésta es todavía una lista en LISP (de seis componentes), así que se puede incluir como parte de una definición ordinaria de función en LISP. Sin embargo, el átomo *IF* no es el nombre de una función primitiva de LISP, de modo que debe ser definida por el programador como función. El programador define IF como una función macro:

$$(define\ IF\ (MACRO\ (X)\ (cuerpo))$$

Cuando el intérprete *apply* de LISP encuentra una llamada *IF* durante la ejecución, invoca la macro *IF* con el argumento completo. El cuerpo de la macro *IF* está definido de tal manera que desarma la lista de entrada y construye la lista:

$$(cond\ ((prueba)(expr_1))(T\ (expr_2)))$$

la cual devuelve como su resultado. Esta lista representa una expresión condicional *cond* válida. *apply* envía entonces esta expresión a *eval* para ser evaluada en lugar de la expresión *IF* original. Ante el programador, parece que la expresión *IF* original ha sido ejecutada de manera directa; la expansión de macro y la evaluación subsiguiente de la expresión *cond* son invisibles.

Gestión de almacenamiento

En casi todas las implementaciones de lenguajes, el ambiente de referencia se mantiene como una estructura de datos definida por el sistema y permanece oculta para el programador. LISP se sale de lo común en cuanto a que hace del ambiente de referencia una estructura de datos que es tanto visible como accesible para el programador. La representación exacta del ambiente de referencia varía ampliamente entre implementaciones de LISP. Las implementaciones con base en compiladores tienden a usar representaciones más bien complejas que permiten referencias eficientes. Sin embargo, la técnica más sencilla y general es la que utiliza una lista de asociaciones explícitas, la lista A. Se describirá esta técnica en seguida.

El área principal de memoria es el montículo, el cual contiene estructuras de listas de datos y también estructuras que representan definiciones de función. Las listas de propiedades se guardan en el montículo, y algunas de las listas definidas por el sistema, como la lista A, también utilizan parte de este almacenamiento. El montículo se maneja en unidades de una palabra de tamaño fijo, usando una lista de espacios libres y el recolector de basura. Una parte del bloque de montículo está restringida a elementos de datos de *palabra completa*, números y cadenas de caracteres, y requiere un plan especial de recolección de basura.

Las otras áreas de memoria incluyen un área asignada estáticamente para programas de sistema, incluido el intérprete, compilador (si se usa), operaciones primitivas y rutinas de gestión de almacenamiento. Una tercera área, llamada *lista de desplazamiento descendente*, está asignada a una pila de registros de activación. Cada registro de activación contiene un punto de regreso y áreas de almacenamiento temporal para ser usadas por una función durante su ejecución. El ambiente de referencia para una función se mantiene en la lista A por separado. La ejecución de una función por el intérprete de LISP, *apply* y *eval*, se describe en las secciones anteriores.

La característica más significativa de la estructura del LISP en tiempo de ejecución es la cantidad de simulación que se necesita. Casi ninguna característica de hardware puede usarse directamente en una computadora convencional; en su lugar, todo debe simularse por software. Cada primitiva está representada por una rutina de software. Las estructuras centrales de control de recursión y *prog* se simulan, e incluso la aritmética se tiene que simular hasta el punto de efectuar verificación de tipos (y posiblemente conversión de tipos) en tiempo de ejecución para cada operación aritmética. La estructura de la computadora virtual en LISP se contrapone casi por completo con el diseño de las computadoras convencionales.

Entorno local de referencia. Una lista A (lista de asociaciones) representa un ambiente de referencia. La lista A es una lista ordinaria de LISP, cada uno de cuyos elementos es un apuntador a una palabra que representa un identificador (átomo) y su asociación vigente. Estas parejas de asociación son palabras de lista cuyos *car* apuntan al átomo y cuyos *cdr* apuntan al valor (lista, átomo u otro objeto de datos) asociado con el átomo.

Las referencias en LISP se hacen estrictamente de acuerdo con la regla de asociación más reciente, la cual se implementa por una simple búsqueda en la lista A de principio (asociaciones más recientes) a fin (asociaciones más antiguas) hasta que se encuentra una asociación adecuada. Así, para encontrar el valor actual de *X*, por ejemplo, se busca en la lista A hasta que se encuentra una entrada cuyo *car* es el átomo *X*. El *cdr* de esa entrada es entonces el valor actual de *X*.

La lista A se modifica durante la ejecución del programa de tres maneras básicas. Cuando ocurre una llamada de función a una función definida por el programador, el intérprete de LISP (función *apply*) aparea los átomos que representan parámetros formales con sus valores correspondientes de parámetros reales y agrega las parejas resultantes al principio de la lista A. Ocurre algo similar cuando se introduce una *prog*. Los átomos que aparecen en la lista como nombres de variables locales para la *prog* se aparean cada uno con el valor *nil* y se agregan al principio de la lista A. Cuando se completa la ejecución de una función o *prog*, se eliminan las asociaciones agregadas al entrar. El programador puede modificar directamente la asociación más reciente para un átomo de la lista A usando *setq* (o *set*). El efecto de (*setq X V AL*) es reemplazar el valor actual de *X* en la lista A por el valor de *V AL*. *X* puede aparecer en la lista A más de una vez, pero *setq* afecta sólo la entrada más reciente.

Ambiente común de referencia. La lista A representa el ambiente local y no local, dinámicamente cambiante, para una función durante su ejecución. También se suministra un ambiente común global en casi todas las implementaciones de LISP. Los detalles de implementación varían ampliamente, pero el método general consiste en permitir la asociación de cualquier átomo con un valor que se utiliza cuando se hace referencia al átomo como un nombre de variable, en vez de una asociación en la lista A. Este valor global, en vez de aparearse con un apuntador al átomo de la lista A, se guarda directamente en el átomo mismo, ya sea en la lista de propiedades del átomo (usando el nombre de atributo *value* o *apval*) o en una localidad especial reservada en la cabeza del átomo para contener un apuntador al valor global.

Cuando se hace referencia a un átomo que tiene un valor global, este valor se recupera en forma directa, sin llevar a cabo una búsqueda en la lista A. Por tanto, un átomo de esta clase no se puede

usar como nombre de parámetro formal o como variable *prog* local, porque cualesquier enlaces con la lista A estarán ocultos en tanto el átomo esté en uso como nombre de variable global. La asignación de un valor a una variable global puede requerir el uso de una función especial de asignación, o se puede usar una función especial para marcar los átomos como nombres de variables globales, empleando *set* y *setq* para asignación en la forma normal.

Paso de parámetros. Los parámetros reales de una llamada de función son siempre expresiones, que se representan como estructuras de listas. LISP suministra dos métodos principales de transmisión de parámetros.

Transmisión por valor. El método más común consiste en evaluar las expresiones de la lista de parámetros reales y transmitir los valores resultantes. Un apuntador a cada valor se aparea con un apuntador al nombre de parámetro formal (átomo) correspondiente y la pareja se inserta en la parte superior de la lista A.

Transmisión por nombre. Un método menos común consiste en transmitir las expresiones de la lista de parámetros reales *sin evaluar*, y dejar que la función llamada las evalúe según se requiera usando *eval*. En este método, la lista completa de parámetros reales se pasa simplemente como un solo parámetro y se aparea en la lista A con el nombre de parámetro formal único de la función llamada. Este método se usa principalmente en dos situaciones: (1) cuando una función acepta un número variable de argumentos en las diferentes llamadas, y (2) cuando puede ser indeseable la evaluación de uno o más de los argumentos (por ejemplo, las primitivas *cond* y *define* requieren que sus argumentos se transmitan sin evaluar). La transmisión por nombre es la norma para funciones macro. Para otras funciones, el programador puede especificar transmisión por nombre usando *nlambda* en lugar de *lambda* en la definición de la función. La definición de función se marca entonces como *fexpr* (funciones interpretadas) o *cfexpr* (funciones compiladas).

Cuando se hace una llamada de función, la función intérprete *apply* verifica el tipo de la función; si es *expr* o *cexpr*, entonces evalúa la lista de parámetros reales antes de llamar la función; si es *macro*, *fexpr* o *cfexpr*, entonces transmite la lista sin evaluar.

Las funciones en LISP pueden crear y devolver objetos arbitrarios de datos como sus valores, en contraste con casi todos los demás lenguajes. Puesto que estos objetos de datos están representados por apuntadores cuando se transmiten como resultados, la transmisión es sencilla. En razón de que una expresión ejecutable se representa como una estructura de listas en tiempo de ejecución, una función puede crear una nueva expresión o definición de función completa y devolver eso como su valor. Por tanto, es posible escribir un programa en LISP que cree o modifique otros programas en LISP.

Se incluyen providencias especiales para transmitir el ambiente de referencia junto con el nombre de la función (cuando la función se transmite como parámetro real a un subprograma) usando la primitiva *function*. El ambiente o entorno de referencia en el punto de transmisión está representado por un apuntador a el tope actual de la lista A. Por ejemplo, para llamar una función *sub* con otra función *nub* como parámetro real, se utiliza la expresión (*sub* (*function nub*)*parámetros*). El átomo *nub* se transmite entonces junto con un apuntador al principio de la lista A vigente, y siempre que se ejecuta *nub* a través de una referencia en *sub* al parámetro formal correspondiente, el apuntador de la lista A transmitida se toma como principio de la lista A que se va a usar como ambiente de referencia durante la ejecución de una *nub*. Este problema general de pasar el ambiente de referencia

durante la ejecución de una nub. Este problema general de pasar el ambiente de referencia correcto junto con un nombre de función cuando se transmite como parámetro se conoce como el *problema funarg* en LISP (funarg = *functional argument* /argumento funcional/).

Funciones normales

Muchas de las funciones siguientes están presentes en casi todas las implementaciones de LISP; sin embargo, sería preferible dar un vistazo a un manual de referencia local para el lenguaje antes de usarlas.

Funciones de lista

(*car L*) devuelve el primer elemento de la lista *L*. (*caar L*) = (*car*(*car L*)), y así sucesivamente.

(*cdr L*) devuelve la lista *L* menos el primer elemento. (*cddr L*) = ((*cdr*(*cdr L*)), y así sucesivamente. *car* y *cdr* se pueden entremezclar, como en (*caddr L*) = (*car*(*cdr*(*cdr L*))).

(*cons x y*) devuelve una lista *L* tal que (*car L*) = *x* y (*cdr L*) = *y*.

(*list x y z*) devuelve la lista (*x y z*).

(*quote x*) o ('*x*) no evalúa *x*.

Predicados

(*atom x*) devuelve cierto si *x* es un átomo.

(*numberp x*) devuelve cierto si *x* es un número.

(*greaterp x y*) devuelve cierto si $x > y$ y (*lessp x y*) devuelve cierto si $x < y$.

(*null x*) devuelve cierto si *x* es el átomo nulo, *nil*.

(*and x y*) devuelve $x \wedge y$.

(*or x y*) devuelve $x \vee y$.

(*not x*) devuelve $\sim x$.

(*eq x y*) devuelve cierto si *x* y *y* son los mismos átomos o las mismas listas. (*equal x y*) devuelve cierto si *x* y *y* son listas con los mismos elementos.

Funciones aritméticas

(+ *x y*) devuelve $x + y$ para los átomos *x* y *y*. Es similar para *, − y /.

(*rem x y*) devuelve el *remainder* (residuo) *de x/y*.

Funciones de entrada y salida

Muchas de estas funciones difieren a través de distintas implementaciones de LISP.

(*load nombredearchivo*) lee e introduce el archivo *nombredearchivo* como una sucesión de definiciones en LISP. Es la interfaz principal con el sistema de archivos para cargar programas.

(*print x*) imprime el elemento *x*. El Scheme también le llama *display*.

(*open nombredearchivo*) abre el archivo *nombredearchivo* y devuelve un apuntador de archivo a él. La manera común de usar esto es guardar el apuntador de archivo a través de una función *setq*:

(setq infile (open '*nombredearchivo*))

y usar la variable *infile* en un enunciado *read*.

(*read*) lee de la terminal el símbolo atómico siguiente (número, carácter o cadena). (*read filename*) lee del apuntador de archivo *nombredearchivo* el símbolo atómico siguiente si se había abierto *filename* previamente.

(*help*) o (*help 'comando*) proporciona informaciòn ùtil.

(trace function_none) rastrea la ejecución de *nombre_de_función* como ayuda para determinar la presencia de errores en un programa.

(*bye*) sale de LISP.

13.1.7 Abstracción y encapsulamiento

El lenguaje de base LISP no incluye características de abstracción de datos; sin embargo, el Common LISP incluye CLOS, el Common LISP Object System (Sistema de Objetos de Common LISP).

CLOS fue el resultado de la fusión de cuatro ampliaciones al LISP orientadas a objetos a mediados de los años ochenta [STEELE y GABRIEL 1993], New Flavors, CommonLoops, Object LISP y Common Objects. CLOS tiene las características siguientes:

1. Se maneja herencia múltiple usando herencia *mixin* (véase la sección 8.2.3).
2. Funciones genéricas.
3. Metaclases y metaobjetos.
4. Una técnica de creación e inicialización de objetos que permite control del proceso por parte del usuario.

13.1.8 Evaluación del lenguaje

LISP, al igual que FORTRAN, es uno de los lenguajes de programación más antiguos. Constituye un pilar de la comunidad dedicada a la inteligencia artificial y ha estado evolucionando durante más de 30 años. Sin embargo, LISP ha sido siempre un lenguaje que ocupa un nicho y no es idóneo para la mayoría de las aplicaciones de computadora convencionales. Aunque las versiones compiladas de LISP permiten que los programas en LISP se ejecuten con un poco más de eficiencia, los avances en el diseño de lenguajes, como el ML de la sección siguiente, suministran una manera más fácil de escribir programas aplicativos. Existen numerosos dialectos de LISP, y los intentos por fusionar los formatos de Scheme y Common LISP en un lenguaje común no han tenido tanto éxito como se quisiera.

Capítulo 14

Lenguajes de programación lógica

14.1 Prolog

Prolog representa el lenguaje principal en la categoría de programación lógica. A diferencia de otros lenguajes de este libro, Prolog no es un lenguaje de programación para usos generales, sino que está orientado a resolver problemas usando el cálculo de predicados. Las aplicaciones del Prolog provienen en general de dos dominios distintos:

- *Preguntas a bases de datos.* Las bases de datos modernas indican típicamente relaciones entre los elementos que están guardados en la base de datos. Pero no todas estas relaciones se pueden indicar. Por ejemplo, en una base de datos de una línea aérea, puede haber entradas que indican números de vuelo, ubicación y hora de salida, y ubicación y hora de llegada. Sin embargo, si un individuo necesita hacer un viaje que requiera cambiar de avión en algún punto intermedio, es probable que esa relación no esté especificada en forma explícita.

- *Pruebas matemáticas.* También se pueden especificar las relaciones entre objetos matemáticos a través de una serie de reglas y sería deseable un mecanismo para generar pruebas de teoremas a partir de este modelo. Aunque la búsqueda ascendente inherente en LISP da cabida a la construcción de sistemas generadores de pruebas, sería eficaz un lenguaje orientado en mayor grado hacia la prueba de propiedades de relaciones.

Estas dos aplicaciones son similares y se pueden resolver usando Prolog.

El objetivo para Prolog era proporcionar las especificaciones de una solución y permitir que la computadora dedujera la secuencia de ejecución para esa solución, en vez de especificar un algoritmo para la solución de un problema, como es el caso normal de casi todos los lenguajes que hemos estudiado. Por ejemplo, si se tiene información de vuelos de una línea aérea de la forma:

vuelo(número_de_vuelo, ciudad_origen, ciudad_destino, hora_de_salida, hora_de_llegada)

entonces todos los vuelos de Los Ángeles a Baltimore se pueden especificar ya sea como vuelos directos:

vuelo(número_de_vuelo, Los Ángeles, Baltimore, hora_de_salida, hora_de_llegada)

o como vuelos con una parada intermedia, especificados como:

```
vuelo(vuelo1, Los Ángeles, X, sale1, llega1),
vuelo(vuelo2, X, Baltimore, sale2, llega2),
sale2 >= llega1+30
```

lo que indica que se está especificando una ciudad X, a la cual llega un vuelo proveniente de Los Ángeles, y de la que sale un vuelo para Baltimore, y el vuelo con destino a Baltimore sale al menos 30 minutos después de que llega el vuelo de los Ángeles para dar tiempo a cambiar de avión. No se especifica un algoritmo; sólo se han indicado las condiciones para tener una solución correcta. El lenguaje mismo, si se puede enunciar un conjunto de condiciones de esta naturaleza, suministrará la secuencia de ejecución que se necesita para encontrar el vuelo apropiado.

14.1.1 Historia

El desarrollo de Prolog se inició en 1970 con Alain Coulmerauer y Philippe Roussel, quienes estaban interesados en desarrollar un lenguaje para hacer deducciones a partir de texto. El nombre corresponde a "PROgramming in LOGic" (Programación en lógica). Prolog fue desarrollado en Marsella, Francia, en 1972. El principio de resolución de Kowalski, de la Universidad de Edimburgo (sección 9.4.4) pareció un modelo apropiado para desarrollar sobre él un mecanismo de inferencia. Con la limitación de la resolución a cláusulas de Horn, la unificación (sección 6.3.2) condujo a un sistema eficaz donde el no determinismo inherente de la resolución se manejó por medio de un proceso de exploración a la inversa, el cual se podía implementar con facilidad. El algoritmo para resolución suministró la secuencia de ejecución que se requería para implementar los ejemplos de especificaciones como la relación de *vuelos* antes señalada.

La primera implementación de Prolog se completó en 1972 usando el compilador de ALGOL W de Wirth, y los aspectos básicos del lenguaje actual se concluyeron para 1973. El uso del Prolog se extendió gradualmente entre quienes se dedicaban a la programación lógica principalmente por contacto personal y no a través de una comercialización del producto. Existen varias versiones diferentes, aunque bastante similares. Aunque no hay un estándar del Prolog, la versión desarrollada en la Universidad de Edimburgo ha llegado a ser utilizada ampliamente. El uso de este lenguaje no se extendió sino hasta los años ochenta. La falta de desarrollo de aplicaciones eficaces de Prolog inhibieron su difusión.

14.1.2 Hola Mundo

Prolog se ejecuta típicamente bajo el control de un intérprete. La función *consult* lee reglas o hechos nuevos hacia la base de datos de Prolog. Si se usa *consult*(*user*), entonces se leen hechos de la terminal, hasta el final de la entrada (que se suele indicar introduciendo *controlD*). *consult*(*nombredearchivo*) lee e introduce hechos desde *nombredearchivo*. La figura 14.1 es nuestro ejemplo "Hola Mundo" en Prolog. La respuesta *yes* (sí) indica que el comando en Prolog tuvo éxito.

```
%prolog
| ?- consult(user).
| writeit :- write('Hola mundo'),nl.
| ^D user consulted, 10 msec 336 bytes
yes
| ?- writeit.
Hola mundo
yes
```

Figura 14.1. Impresión de "Hola Mundo" en Prolog.

14.1.3 Breve perspectiva del lenguaje

Un programa en Prolog se compone de una serie de hechos, relaciones concretas entre objetos de datos (hechos) y un conjunto de reglas, es decir, un patrón de relaciones entre los objetos de la base de datos. Estos hechos y reglas se introducen en la base de datos a través de una operación de consulta.

Un programa se ejecuta cuando el usuario introduce una pregunta, un conjunto de términos que deben ser todos ciertos. Los hechos y reglas de la base de datos se usan para determinar cuáles sustituciones de variables de la pregunta (llamadas unificación) son congruentes con la información de la base de datos.

Como intérprete, Prolog solicita entradas al usuario. El usuario digita una pregunta o un nombre de función. La verdad ("yes") o falsedad ("no") de esa pregunta se imprime, así como una asignación a las variables de la pregunta que hacen cierta la pregunta, es decir, que *unifican* la pregunta. Si se introduce un ";", entonces se imprime el próximo conjunto de valores que unifican la pregunta, hasta que no son posibles más sustituciones, momento en el que Prolog imprime "no" y aguarda una nueva pregunta. Un cambio de renglón se interpreta como terminación de la búsqueda de soluciones adicionales.

La ejecución de Prolog, aunque se basa en la especificación de predicados, opera en forma muy parecida a un lenguaje aplicativo como LISP o ML. El desarrollo de reglas en Prolog requiere el mismo "pensamiento recursivo" que se necesita para desarrollar programas en esos otros lenguajes aplicativos.

Prolog tiene una sintaxis y semántica simples. Puesto que busca relaciones entre una serie de objetos, la variable y la lista son las estructuras de datos básicas que se usan. Una regla se comporta en forma muy parecida a un procedimiento, excepto que el concepto de unificación es más complejo que el proceso relativamente sencillo de sustitución de parámetros por expresiones (véase la sección 6.3.2).

Prolog emplea la misma sintaxis que C y PL/I para comentarios: /* ... */.

```
1  %editor data.prolog
2        /* Leer en datos como una Relación de Prolog */
3        valsdedatos(4[1,2,3,4]).
4  %editor pgm.prolog
5         go :- reconsult('data.prolog'),
6               valsdedatos(A,B),
7               SUMAEN is 0,
8               for(A,B,SUMAEN,SUMASAL),nl,
9               write('SUMA ='),write(SUMASAL),nl.
10       /* El lazo for se ejecuta 'I' veces */
11       for(I,B,SUMAEN,SUMASAL) :- not(I=0),
12              B=[CABEZA|COLA],
13              write(CABEZA),
14              VALNUEVO is SUMAEN+CABEZA,
15              for(I-1,COLA,VALNUEVO,SUMASAL).
16       /* Si I es 0, devolver el valor calculado de SUMAEN */
17       for(_,_,SUMAEN,SUMASAL) :- SUMASAL = SUMAEN.
18       not(X) :- X, !, fail.
19       not(_).
20 %prolog
21 | ?- consult('pgm.prolog').
22 {consulting /aaron/mvz/book/pgm.prolog...}
23 {/aaron/mvz/book/pgm.prolog consulted, 30 msec 1456 bytes}
24 yes
25 | ?- go.
26 {consulting /aaron/mvz/book/data.prolog...}
27 {/aaron/mvz/book/data.prolog consulted, 10 msec 384 bytes}
28 1234
29 SUMA= 10
30 yes
```

Figura 14.2. Ejemplo en Prolog para sumar un arreglo.

Ejemplo anotado

En este ejemplo, como en el caso de los lenguajes anteriores, se muestra un programa en Prolog que suma un vector de números. Para conservar la sencillez del problema, se supondrá que los datos de entrada tienen la forma de una relación en Prolog: *valsdedatos*(*elementos_número*, [*elementos_lista*]). La figura 14.2 muestra la ejecución de este programa.

Renglones 1-3. Estos invocan el editor para crear el archivo de datos. La entrada será un hecho `valsdedatos`, el cual contiene una cuenta y una lista de números por sumar. La cuenta es opcional, puesto que el programa se podría haber escrito de modo que contara el número de elementos de la lista.

Renglón 4. Éste invoca el editor para crear el programa. El programa se podría haber digitado directamente a Prolog desde el teclado a través de una `consult(user)` en vez de una `consult('pgm.prolog')` en el renglón 21.

Renglón 5. Éste define el objetivo principal del programa, la resolución del predicado `go`. `go` se resuelve si los renglones del 5 al 9 se resuelven. `go` vuelve a consultar primero la base de datos leyendo el archivo `data.prolog` y agrega los hechos a la base de datos, en este caso, el hecho `valsdedatos` del renglón 3.

Renglón 6. Éste tiene el efecto de asignar (de hecho, unificar) `A` al primer elemento de valsdedatos, la cuenta de elementos (`A = 4`) y `B` al vector de elementos (`B` = [1,2,3,4]).

Renglón 7. SUMAEN se fija en 0. `is` se requiere, = significa identidad, y simplemente unifica ambos objetos de modo que sean iguales si es posible.

Renglón 8. Este renglón crea un lazo 'for' usando recursión. La regla se ejecuta `I` veces (`I` es el número de elementos en el vector B), pasa hacia adentro el vector `B` y la suma actual `SUMAEN` y devuelve `SUMASAL = SUMAEN + elementos de B`. Puesto que la regla `for` imprime cada valor de dato, `nl` imprime un carácter de renglón nuevo.

Renglón 9. Al regresar de la regla `for`, `SUMASAL` tiene la suma deseada. Las dos funciones `write` imprimen el valor deseado.

Renglón 10. Los comentarios en Prolog son como los comentarios en C y se pueden colocar en cualquier punto.

Renglones 11-17. Éstos definen la regla `for`. La semántica de la regla consiste en ejecutar la regla `I` veces. Si `I` no es 0, entonces esta regla continúa ejecutándose; si fracasa, entonces el retroceso de Prolog ejecutará la segunda regla `for` del renglón 17.

Renglón 12. Separa el vector de entrada en un componente CABEZA (primer elemento) y un componente COLA (residuo).

Renglón 13. Se imprime el valor del primer elemento.

Renglón 14. Los elementos de vector previamente observados y el elemento vigente se suman para calcular la suma inicial siguiente.

Renglón 15. La nueva suma parcial y el vector COLA acortado se pasan a la regla for una vez más en forma recursiva, con la cuenta disminuida en 1.

Renglones 16-17. El renglón 16 es un comentario. El renglón 17 es la segunda regla `for` que se ejecuta si fracasa la primera regla `for`. La primera fracasará si `not (I=0)` fracasa (es decir, `I` es 0). En este caso, no se toman en cuenta los dos primeros argumentos y simplemente se unifica la suma del arreglo (`SUMASAL`) con el valor que se ha estado calculando (`SUMAEN`). Obsérvese que si se invierte el orden de las dos reglas `for` de los renglones 11-15 y 17, entonces esta segunda regla se ejecutará primero en todos los casos y el programa no funcionará como se desea.

Renglones 18-19. Ésta es la definición normal de `not` según se explica en el texto. Si `X` es cierto en la base de datos, entonces se ejecuta la `fail`. El corte `!` impide el retroceso, así que `not` fracasa. Si `X` no es cierto, entonces la primera regla `not` fracasa, pero se permite retroceso, puesto que todavía no se ha ejecutado el corte, y el segundo se ejecuta en el renglón 19. Esta regla siempre tiene éxito.

Renglón 20. Ahora que se han creado los archivos de datos y de programa, se invoca Prolog.

Renglón 21. Lee en las reglas desde el archivo que tanto `'pgm.prolog'. consult(user)` como las reglas podrían haberse ecrito directamente en el programa.

Renglones 22-24. Ésta es la salida de Prolog que dice que el archivo se leyó satisfactoriamente.

Renglón 25. La entrada go. significa una pregunta, definida en los renglones 5-9, la cual define el éxito para el programa.

Renglones 26-27. Salida que produce Prolog como resultado de leer el archivo de datos en el renglón 6.

Renglones 28-29. Salida deseada que produce el programa.

Renglón 30. El go del renglón 25 tiene éxito. La salida desde Prolog suele ser un indicador de final de archivo (por ejemplo, *controlD*).

14.1.4 Objetos de datos

Tipos de datos primitivos

Variables y constantes. Los datos en Prolog incluyen enteros (1, 2, 3, ...), reales (1.2, 3.4, 5.4, ...) y caracteres ('a', 'b', ...). Los nombres que comienzan con letra minúscula representan hechos concretos, si son parte de un hecho de la base de datos, o, si están escritos con mayúsculas, representan variables que se pueden unificar con hechos durante la ejecución de Prolog. El alcance de una variable es la regla dentro de la cual se usa.

Tipos de datos estructurados

Los objetos pueden ser átomos (constantes y variables de cadena) o listas. Una lista se representa como [*A*,*B*,*C*,...]. La notación [*A*|*B*] se usa para indicar *A* como cabeza de la lista y *B* como cola de la lista:

```
?- X = [A|B], X=[1,2,3,4]. – Preguntar 2 definiciones de X
                  –Imprimir unificación de A, B y X
A = 1,
B = [2,3,4],
X = [1,2,3,4]
```

La cadena *'abcd'* representa la lista [*'a','b','c','d'*].

La variable _ es una variable arbitraria sin nombre. Así, [*A* | _] hace coincidir *A* con la cabeza de cualquier lista.

Tipos definidos por el usuario

No existen verdaderos tipos definidos por el usuario; sin embargo, las reglas para definir relaciones pueden actuar como si fueran tipos de usuario.

Representación de almacenamiento

Las reglas y hechos se guardan como listas vinculadas en la memoria. La ejecución básica de Prolog es similar a la estructura de recorrido de árbol que se describió para el LISP.

14.1.5 Control de secuencia

Dada una pregunta, Prolog emplea unificación con retroceso, como se describe en las secciónes 6.3.2 y 6.3.3, como mecanismos para administrar el almacenamiento. No existe un auténtico concepto de ambientes locales o globales. Todas las reglas tienen contexto local.

A causa de la manera como se ha implementado la unificación, una pregunta:

$$q_1, q_2, \cdots q_n$$

evalúa primero q_1, luego q_2, y así sucesivamente. Esto tiene la apariencia de un orden secuencial para cualquier regla. Además, las reglas se guardan en el orden en que fueron introducidas en la base de datos. Así, *not* se puede definir como:

```
not(X) :–X, !, fail.
not(_).
```

en vez de:

```
not(_).
not(X) :–X, !, fail.
```

En el segundo caso, *not*(_) se verificaría primero y siempre tendría éxito.

Expresiones

Se definen las operaciones aritméticas +, – *, *mod* y /. También se definen relaciones tales como =, =< (adviértase que ésta no se escribe <=), <, > y >=. Pero el operador de igualdad significa "lo mismo que", así que $X = 1 + 2$ significa que X es lo mismo que la expresión 1+2. El operador *is* significa evaluar; *X is* 4+1 da por resultado una asignación de 5 a X:

```
| ?– X = 1+2
X = 1+2? – X  expresión asignada '1+2'
yes
| ?– X is 1+2
X = 3? – X valor asignado 3
yes
```

A causa de este significado del operador =, las expresiones como 1 + 3 = 2 + 2 son falsas. Para evaluar éstas, es necesario definir una función de igualdad que utilice la construcción *is* para forzar la evaluación de ambos términos:

```
mismovalor(X,Y) :–A is X, B is Y, A=B.
```

Entonces *mismovalor*(1 + 4, 2 + 3) devolverá cierto.

Enunciados

Hechos. Los hechos son relaciones que se expresan durante una operación *consult*, la cual agrega nuevos hechos y reglas a la base de datos. Son tuplas n $f(a_1,a_2,...,a_n)$ y expresan la relación entre los n argumentos de acuerdo con la relación *f*. En respuesta a la función que agrega datos a la base de datos, *consult*(*user*), el usuario podría introducir:

```
patrón(zelkowitz,universidad).
patrón(smith,nasa).
patrón(pratt,nasa).
editor(zelkowitz,prenticehall).
editor(pratt,prenticehall).
controlD
```

Estas relaciones son invariables; no se pueden unificar a algún otro valor.

Reglas. Las reglas son implicaciones que se expresan durante una operación *consult*. Tienen la sintaxis:

```
empleado(X) :–patrón(X,_). /* X está empleado si tiene un patrón */
empleadopor(X) :–patrón(X,Y), write(Y).
```

empleado sólo verifica que *patrón*(*X*,*algo*) esté en la base de datos, de modo que no se toma en cuenta el patrón, pero *empleadopor*(*X*) usa la relación *patrón*(*X*,*Y*) para escribir el nombre del patrón:

```
empleado(pratt).
yes
empleadopor(zelkowitz).
universidad
yes
```

Preguntas. Las preguntas consisten en una sucesión de términos que terminan con un punto.

$$q_1, q_2, \cdots, q_n$$

Se desea que una asociación para las variables de cada término sea de tal naturaleza que todos los términos sean ciertos. La ejecución tiene lugar unificando q_1, posiblemente haciendo referencia a su definición en la base de datos y, si es cierta, unificando q_2 con su definición, etc. Si la unificación fracasa en cualquier punto, el proceso retrocede a la elección correcta anterior e intenta una alternativa, como se ha indicado en la sección 6.3.2.

Las reglas se pueden combinar para hacer preguntas complejas. Por ejemplo, la pregunta "¿Quién trabaja para la NASA y tiene un libro publicado por Prentice Hall?" se puede expresar como la pregunta:

```
patrón(X,nasa),editor(X,prenticehall).
X = pratt ?
yes
```

Aunque tanto 'zelkowitz' como 'pratt' unifican *editor(X, prenticehall)*, sólo 'pratt' unifica con *patrón(X, nasa)*.

Cortes. El proceso de evaluación de preguntas suele requerir considerable retroceso. Para ahorrar tiempo, se introdujo el *corte*. Un corte (!) indica que si se debe aplicar retroceso, el patrón fracasa. El efecto de un corte es limitar el espacio de búsqueda para una solución. El uso del corte nunca agregará una solución que no se habría podido encontrar sin un corte, pero puede eliminar ciertas soluciones válidas.

Por ejemplo, si la pregunta anterior se hubiera escrito como:

```
patrón(X,nasa),!,editor(X,prenticehall).
```

entonces la pregunta fracasaría. *patrón(smith,nasa)* unificaría primero, pero *smith* no conseguiría unificar con *editor(X,prenticehall)*. Normalmente, Prolog retrocedería y buscaría otra relación *patrón(X,nasa)*, pero el corte indica que no se haga ese retroceso. El corte no tiene efectos en la dirección de avance.

Problemas con la negación. Not se define como:

```
not(X) :–X, !, fail.
not(–).
```

Adviértase que esto no es lo mismo que *devolver cierto si X es falso*. Si *X* es cierto, entonces *not(X)* evaluará *X* como cierto, fracasará en *fail*, pero el corte fuerza el fracaso de la regla. Si *X* es falso, entonces la primera regla fracasa, pero *not(–)* tiene éxito. La diferencia se puede ver en las dos preguntas siguientes:

```
X is 5, not(X=10)
not(X=10), X is 5
```

En el primer caso, *X* unifica con 5 y not(X=10) tiene éxito, pero, en el segundo caso, *X* unifica primero con 10 y fracasa, de modo que la unificación con 5 nunca se ejecuta.

Entrada y salida

Para muchas consultas sencillas, la salida por omisión de las variables unificadas es suficiente para escribir respuestas. Sin embargo, también existe una función *write* para escribir cualquier cadena. *write('abc')* imprime *abc*. La función *nl* escribe un símbolo de nuevo renglón.

14.1.6 Subprogramas y gestión de almacenamiento

Prolog tiene dos modos: modo de consulta y modo de pregunta. En el modo de consulta se introducen nuevas relaciones en el almacenamiento dinámico de la base de datos. Durante el modo de pregunta, se ejecuta un intérprete simplemente basado en pilas para evaluar preguntas, como se indicó en la sección 6.3.2.

Ambiente local de referencia. Todos los nombres de variables son locales para la regla en la cual están definidos. La unificación prevé la interacción de nombres locales de una regla con los nombres de otra regla.

Ambiente común de referencia. Todos los datos son compartidos. No existe un verdadero concepto de ambiente local o global de referencia.

Paso de parámetros. La unificación prevé el paso de argumentos entre reglas.

Funciones normales

Casi todos los sistemas en Prolog incluyen varias funciones integradas para ayudar en la generación de programas:

consult(*nombredearchivo*) lee el archivo *nombredearchivo* y anexa nuevos hechos y reglas a la base de datos. *nombredearchivo* también se puede escribir como *'nombredearchiv'* si el nombre del archivo contiene caracteres incrustados no pertenecientes a un identificador. *consult*(*nombredearchivo*) se suele poder especificar simplemente como [*nombredearchivo*].

reconsult(*nombredearchivo*) sobreescribe relaciones en la base de datos.

fail siempre fracasa.

see(*nombredearchivo*) lee entradas desde *nombredearchivo* como una serie de reglas.

write(*término*) escribe *término*.

tell(*nombredearchivo*) reorienta la salida de *write* a *nombredearchivo*.

told cierra el archivo respecto a una *tell* previa y reorienta la salida de *write* a la terminal normal de visualización.

nl pasa al renglón siguiente (de entradas y salidas).

atom(*X*) es un predicado que devuelve cierto si X es un átomo (constantes de cadena o nombres de variable).

var(*X*) es un predicado que devuelve cierto si *X* es una variable.

integer(*X*) es un predicado que devuelve cierto si *X* es un entero.

trace activa la depuración por rastreo mostrando cada paso de la ejecución. *notrace* la desactiva.

14.1.7 Abstracción y encapsulamiento

Prolog no proporciona medios para verdaderas capacidades de abstracción.

14.1.8 Evaluación del lenguaje

Prolog ha alcanzado una medida razonable de éxito como lenguaje para resolver problemas de relaciones, como en el procesamiento de consultas a bases de datos. Ha alcanzado un éxito limitado en otros dominios.

El objetivo original de poder especificar un programa sin proporcionar sus detalles algorítmicos no se ha alcanzado realmente. Los programas en Prolog "se leen secuencialmente", aunque el desarrollo de reglas sigue un estilo aplicativo. Se usan cortes con frecuencia para limitar el espacio de búsqueda para una regla, con el efecto de hacer el programa tan lineal en cuanto a ejecución como un programa típico en C o Pascal. Aunque las reglas se suelen expresar en forma aplicativa, el lado derecho de cada regla se procesa de manera secuencial. Esto sigue un patrón muy similar al de los programas en ML.