



BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Conceptos de lenguajes de programacion

Carlo Ghezzi y Mehdi Jazayeri



BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Conceptos de lenguajes de programación

Traducido por:

F. Aylagas, P. Carazo, E. Pérez,
J. M. Moreno, J. C. Galilea y J. A. Valverde

Ediciones Díaz de Santos, S. A.



BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Donación Facultad

Fecha 24-10-2013

Inv. DIF - FO57

D.3
GAE

DONACION H. Ghezzi Zegarra

\$.....

Fecha 16-5-05

Inv. E..... Inv. B..... 1457

J.E
GHE

1457

Copyright © 1982. John Wiley & Sons Inc.
Copyright © 1986. Diaz de Santos, S.A.

"No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright".

ISBN: 84-86251-40-0

Depósito Legal: M-26.154-1986

Edita: DIAZ DE SANTOS, S. A.
c/. Juan Bravo, 3-A
28006 MADRID

Portada ASEL
Imprime: Imp. CALERO-MADRID



BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

C O N C E P T O S
D E L E N G U A J E S
D E P R O G R A M A C I O N

Carlo Ghezzi
Mehdi Jazayeri



BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

PROLOGO

HUMBERTO CHAVEZ ZEGARRA
LIMA - PERU

El lenguaje sólo es un instrumento de la ciencia, y las palabras no son más que expresiones de las ideas.
Samuel Johnson (1709-1794)

Este libro presenta, analiza y evalúa los conceptos más importantes que suelen aparecer en los lenguajes de programación actuales. Está basado en dos premisas: que el propósito de los lenguajes de programación es la producción de programas, y que estos sólo son una de las muchas herramientas requeridas para tal tarea. El valor de un lenguaje, o de un concepto del lenguaje, se debe juzgar según el modo en que afecta a la producción de software y dependiendo de la facilidad con que se puede integrar con otras herramientas.

El presente libro no es una introducción a ningún lenguaje de programación, ni tampoco es un examen exhaustivo de ellos, sino que presenta y evalúa conceptos comunes a los lenguajes, así como aquellos que seguramente van a marcar la evolución de los mismos. Por este motivo hemos elegido un enfoque comparativo, es decir, los conceptos se muestran contrastando su aparición en los diferentes lenguajes. Los lenguajes en los que se ha hecho más hincapié han sido Pascal, ALGOL 68 y SIMULA 67, ya que: 1) han proporcionado el punto de partida para muchas ideas de los actuales lenguajes, y 2) muestran con claridad las características de los lenguajes primitivos. Muchos otros, desde los más antiguos, como FORTRAN y ALGOL 60, hasta los más recientes, como CLU, Euclid, Mesa y Ada, se utilizan como ejemplos, con cierto énfasis particular en Ada. Un capítulo está dedicado a los lenguajes funcionales, prestando especial atención a LISP y APL. De cualquier manera, el lector no se convertirá en un eficiente programador en ninguno de estos lenguajes. En su lugar, esperamos que este libro le ayude a mejorar su habilidad para apreciar y evaluar lenguajes de programación. Además, esperamos que sea capaz de identificar los conceptos más importantes de los lenguajes y reconocer sus límites y posibilidades.

Esta habilidad que hemos mencionado es esencial si se va a elegir un lenguaje, o bien a diseñarlo, o incluso para utilizarlo aprovechando al máximo sus ventajas. Esperamos pues, que el libro demuestre su utilidad para los coordinadores de proyectos, programadores profesionales y diseñadores de lenguajes, tanto en la faceta de guía de estudio, como en la de texto de referencia.

La labor de los profesionales se ha vuelto más difícil con la rápida explosión de información en el campo de los lenguajes de programación, a la vez que se ha reconocido que nuestros lenguajes actuales son inadecuados. Esta tendencia está claramente demostrada con el creciente número de conferencias sobre lenguajes, por el incremento en el número de lenguajes para microprocesadores, debido a los avances en la tecnología del hardware, y por la decisión del Departamento de Defensa de los Estados Unidos de patrocinar el desarrollo de un lenguaje más. Nuestra intención es permitir al lector beneficiarse de estos

desarrollos en lugar de verse desbordado por ellos.

Hemos diseñado el libro para que se pueda usar como texto en un curso sobre lenguajes de programación. En tal caso, se debería complementar con los manuales específicos de cada lenguaje. Hemos tomado la decisión deliberada de no incluir en el texto ningún "resumen" de ningún lenguaje. Un resumen puede proporcionar solamente una visión superficial, y por lo tanto no se puede utilizar como documento de referencia; sólo el manual del lenguaje o su definición oficial pueden servir para este propósito. Es más, uno de los principios básicos que un estudiante debe adquirir en un curso de este tipo, es la habilidad para aprender y evaluar un lenguaje basándose solamente en su definición oficial. El Glosario de Lenguajes que se encuentra al final del libro encamina al lector a los documentos necesarios. De cualquier modo, el libro es en sí un análisis de conceptos de lenguajes de programación.

Para la lectura del presente libro se requiere fluidez en un lenguaje y conocimiento con experiencia de otro, preferiblemente alguno de ellos con estructura de bloques. El conocimiento (simplemente de lectura) de Pascal es útil pero no esencial. También sería conveniente que el lector tuviera alguna experiencia en el desarrollo de software. El énfasis en el diseño sistemático de programas requiere que el lector esté familiarizado con los problemas y dificultades que entraña la producción de software. Asimismo, deberá ser consciente, en particular, de la necesidad de un enfoque disciplinado de la programación, y deberá saber cómo usar un lenguaje de una manera disciplinada. Aunque esto último no es necesario si el lector está dispuesto a creerse, con buena fe, algunos de nuestros asertos.

El énfasis en lenguajes modernos, como Pascal, en lugar de los tradicionales, como FORTRAN o PL/I, hace que este libro esté especialmente recomendado para universidades que comparten este énfasis. Aunque la mayoría de las tendencias van en esa dirección, también tratamos las bases de los lenguajes tradicionales.

Algunas partes del libro se han venido usando, durante los últimos cinco años, en cursos universitarios de iniciación a lenguajes. El material aquí contenido es apropiado tanto para estudiantes de este nivel como para los de un nivel más avanzado.

OBJETIVO Y CONTENIDO

Si en algún modo se distingue este trabajo de otros libros sobre lenguajes de programación, es en el hecho de que los lenguajes deben soportar el desarrollo de software. En realidad, uno de los objetivos consiste en evaluar conceptos de lenguajes en base a su contribución al proceso de desarrollo de software y al estudio de los criterios necesarios para tal evaluación. Estos criterios se basan principalmente en la relación entre los

lenguajes, de programación y los métodos de diseño, y aunque en menor grado, en la relación entre los lenguajes y otras ayudas para la producción de software. Este punto de vista se trata en el Capítulo 1. En el Capítulo 2 presentamos cuatro conceptos: tipos de datos, estructuras de control, corrección del programa y programación a gran escala, los cuales se pueden utilizar para examinar y evaluar los lenguajes. Estos conceptos también se emplean para explicar la evolución del tema que nos ocupa, a través de los 25 últimos años. Cada uno de los Capítulos 4 a 7 están dedicados a detallar e ilustrar cada uno de estos cuatro conceptos citados. Un punto importante de estos capítulos es el modo en que se pueden implementar estos conceptos con las características del lenguaje. El Capítulo 3 proporciona la base necesaria para poder seguir las debates de los capítulos 4 a 7.

Aunque nos dedicamos principalmente a los lenguajes clasificados como imperativos, u orientados a la sentencia, el Capítulo 8 se ocupa de los lenguajes funcionales o aplicativos. El contraste entre estos dos estilos, imperativo y aplicativo, sirve para mostrar su potencia y sus defectos.

Por último, en el Capítulo 9 tratamos el tema del diseño de un lenguaje. Este campo ha sido, y promete continuar siéndolo, una de las áreas más activas de la ciencia de los ordenadores. El reciente lenguaje Ada, del Departamento de Defensa de los Estados Unidos, y muchos otros lenguajes para microordenadores que se están desarrollando hoy día, indican este alto nivel de actividad.

AGRADECIMIENTOS

Los comentarios de ciertas personas han servido de gran ayuda para reducir los errores y mejorar la presentación. En especial, queremos dar las gracias a J. P. Banatre, Dan Berry, Lawrence Chan, Jon Cohen, Richard Leblanc, Gyula Magó, Dino Mandrioli, David Moffat y a Kuo-Chung-Tai.

Carlo Ghuezzi y Mehdi Jazayeri agradecen la ayuda del CNR (Junta Italiana de Investigación) y del TRW Vidar.

Fuimos muy afortunados al contar con la colaboración de Edith Coon, que mecanografió complaciente, precisa y rápidamente este texto. Por las muchas versiones que ha escrito y vuelto a escribir, le damos las gracias. Vicki Baker mecanografió las primeras versiones de varios capítulos. También queremos dar las gracias a nuestras familias que nos animaron en las muchas horas dedicadas al "libro".

NOTA AL PROFESOR

Los Capítulos 2 al 9 contienen ejercicios. Unos cuantos requieren que el estudiante consulte las definiciones oficiales de los lenguajes. Hay dos tipos de ejercicios que aunque no se

citan explícitamente, son de particular interés. Uno consiste en la lectura del manual de referencia de un lenguaje y la evaluación de tal lenguaje basándose en los conceptos descritos en este libro. El Glosario de Lenguajes, al final del libro, puede servir como punto de partida. El otro ejercicio que recomendamos es la escritura de programas en diferentes lenguajes. Es posible que en una instalación particular no se disponga de los compiladores de los lenguajes aquí tratados. Lo que recomendamos en tal caso, es que se utilicen estos lenguajes para diseñar programas cuando sea conveniente (p.ej. CLU para diseñar un tipo de dato abstracto) y a continuación traducir manualmente la solución a un lenguaje del que se tenga compilador. Este tipo de ejercicio ayuda al estudiante a apreciar la utilidad de los lenguajes que soportan directamente un método de diseño.

El Capítulo 7 trata acerca de herramientas de desarrollo de software, haciendo énfasis en UNIX. Si UNIX está disponible en su instalación, recomendamos al estudiante trabajos de programación que le hagan utilizar las diferentes herramientas de desarrollo. Ello podría causar un verdadero cambio de actitud hacia la programación, especialmente si se está acostumbrado a escribir programas en un entorno poco grato (p.ej. utilizando tarjetas perforadas, comandos JCL (de control de trabajos) cripticos, etc.). También, para ayudar a los estudiantes a apreciar los problemas de la programación a gran escala, se deberían formar equipos para proyectos relacionados con las ideas desarrolladas en el Capítulo 7.

Carlo Ghezzi

Mehdi Jazayeri

NOTA DE LOS TRADUCTORES

Hace aproximadamente año y medio cayó en nuestras manos uno de los primeros ejemplares de la versión original de este libro. No nos fue difícil darnos cuenta desde un primer momento de la extraordinaria calidad del mismo. No es usual en absoluto encontrarse con libros capaces de explicar temas de un cierto nivel informático, con la claridad y amplitud con que sus autores lo han logrado en esta ocasión. Nuestra doble condición de profesionales de la Informática por un lado, y de docentes de la misma por otro, nos hizo valorar aún más el trabajo realizado en la preparación del original. Y de una manera bastante natural surgió la idea de que un texto así no debía quedar reducido a ciertos ambientes profesionales, sobre todo, dada la escasez de buen material de aprendizaje, tanto en entornos académicos, como en todos los ambientes interesados en la Ingeniería de la Programación, dentro o fuera de un entorno industrial.

Y con la mejor de nuestras voluntades acometimos la labor de traducción del texto inglés, no sin antes haber hecho todo tipo de esfuerzos para lograr una unificación de criterios, que parecía necesaria dada la pluralidad en el número de traductores.

Por otro lado, al acometer hoy en día en España una empresa de este tipo, caben dos opciones: realizar una semi-traducción, dejando sin traducir todos aquellos términos técnicos que ya tienen un significado claro para todos los profesionales, o contribuir con nuestro grano de arena a detener esta invasión de terminología sajona que día a día empobrece nuestra lengua. Nosotros hemos optado por esta segunda vía, a sabiendas de la dificultad que conlleva, si bien es cierto que en un principio no supimos calibrar la cantidad de trabajo que esta decisión nos iba a suponer.

Hemos tratado en todo momento de encontrar, cuando todavía no estaba encontrada, la palabra o expresión castellana que mejor recogía la esencia del término original inglés. Solamente hemos renunciado a hacerlo en aquellos casos en que la expresión original contiene matices muy difíciles o imposibles de expresar de una forma concreta en castellano ("cluster"), o en aquellos otros en que una traducción convertiría el texto castellano en algo realmente ilegible, dada la amplia aceptación del vocablo sajón ("software"). En todo caso, en la primera aparición de todo vocablo traducido de una forma que podría crear cierta confusión a un lector habituado a la utilización de bibliografía sajona, hemos mantenido la versión original entre paréntesis, con el fin de guiarle en la lectura posterior.

Pensamos que el beneficio que de esta decisión puede obtener el lector novel, supera con mucho el pequeño perjuicio que pueda suponer para el lector avezado.

No es fácil ahora, a todo pasado, evaluar la cantidad de reuniones que entre los miembros del equipo de traducción han merecido esta o aquella expresión, tal o cual palabra (y eso que todos procedemos de una raíz académica y profesional común). Aún así, somos conscientes de que en muchos casos no habremos acertado, y que innumerables lectores del libro pensarán que su propia traducción sería mejor, o que no les gusta el matiz empleado. No obstante, creemos que lo realmente importante es que entre todos contribuyamos a aclarar y castellanizar un poco la jerga que nos invade.

Y como siempre, queremos agradecer su colaboración a todas aquellas personas de nuestro entorno que se han visto asediadas alguna vez para poner paz en nuestras discusiones o en nuestras dudas, dándonos su siempre bien intencionada opinión. Con el riesgo de que se nos olvide alguien no queremos dejar de citar a Pilar Manzano, Angeles Hermosa, M. Dolores Hinojal, Angel Pérez Riesco, Cristina Sánchez, Gonzalo Sánchez Dueñas y M. José Membrillera. Tampoco queremos dejar de reconocer las facilidades que nos han proporcionado las empresas Secoinsa e Icuatro a la hora de utilizar equipos suyos para la edición del presente texto.

El equipo traductor

Francisco Aylagas
 Pablo Carazo
 Juan Carlos Galilea
 Juan Manuel Moreno
 Eduardo Pérez
 Juan Antonio Valverde



BIBLIOTECA
 FAC. DE INFORMÁTICA
 U.N.L.P.

ÍNDICE

	Págs.
1. INTRODUCCION	
1.1 El Proceso de Desarrollo de los Programas	27
1.2 El Lenguaje como un Componente más de una Utilidad Global para el Desarrollo de Programas	29
1.3 Métodos para el Desarrollo de Programas y Lenguajes de Programación	31
1.4 La Arquitectura del Ordenador y los Lenguajes de Programación	32
1.5 Objetivos en el Diseño de Lenguajes Impuestos por el Proceso de Desarrollo de los Programas	33
1.5.1 Lenguaje y Fiabilidad	34
1.5.2 Lenguaje y Mantenibilidad	35
1.5.3 Lenguaje y Eficiencia	36
1.6 Una Breve Perspectiva Histórica	36
2. EVOLUCION DE CONCEPTOS EN LOS LENGUAJES DE PROGRAMACION	
2.1 El Papel de la Abstracción	44
2.2 Las Abstracciones de Datos	46
2.2.1 Abstracción de Datos en los Lenguajes Primitivos	46
2.2.2 Abstracción de Datos en ALGOL 68, Pascal y SIMULA 67	47
2.2.3 Hacia los Tipos Abstractos de Datos	49
2.3 Abstracción de Control	51
2.3.1 Evolución de las Estructuras de Control a Nivel de Sentencia	51
2.3.2 Evolución de las Estructuras de Control a Nivel de Unidad de Programa	52
2.3.2.1 Subprogramas y Bloques	52
2.3.2.2 Manejo de Excepciones	53
2.3.2.3 Corrutinas	54
2.3.2.4 Unidades Concurrentes	54

18. Conceptos de Lenguajes de Programación

Págs.

2.4 Corrección	58
2.5 Programación a gran Escala	61
3. INTRODUCCION ALA SEMANTICA DE LOS LENGUAJES DE PROGRAMACION	
3.1 Proceso de un Lenguaje	70
3.1.1 Interpretación	70
3.1.2 Traducción	71
3.2 Concepto de Ligadura	72
3.3 Variables	73
3.3.1 Ambito de una Variable	73
3.3.2 Tiempo de Vida de una Variable	74
3.3.3 Valor de una Variable	74
3.3.4 Tipo de una Variable	76
3.4 Unidades de Programa	79
3.5 Estructura de FORTRAN	80
3.6 Estructura de Lenguajes tipo ALGOL	82
3.6.1 Registro de Activación de Tamaño Conocido Estáticamente	84
3.6.2 Registros de Activación cuyo Tamaño es Conocido en la Activación de la Unidad	87
3.6.3 Registros de Activación de Tamaño Variable Dinámicamente	89
3.6.4 Acceso al Entorno Global	90
3.6.4.1 Cadena estática	91
3.6.4.2 Display	94
3.7 Estructura de un Lenguaje Interpretativo: APL	96
4. TIPOS DE DATOS	
4.1 Tipos Predefinidos	105
4.2 Agregados de Datos	108

4.2.1 Producto Cartesiano	108
4.2.2 Aplicación Finita	109
4.2.3 Secuencia	110
4.2.4 Recursión	111
4.2.5 Unión Discriminada	112
4.2.6 Conjunto Potencia	112
4.3 Tipos Definidos por el Usuario	113
4.3.1 Estructura de Tipos de ALGOL 68	115
4.3.1.1 Tipos Primitivos (Modos Sencillos)	116
4.3.1.2 Tipos no Primitivos (Modos Complejos)	116
4.3.2 Estructura de Tipos de Pascal	125
4.3.2.1 Tipos no Estructurados	125
4.3.2.2 Constructores de Agregados	127
4.4 Conversión de Tipo	135
4.5 Áreas Problemáticas y Soluciones Sencillas	137
4.5.1 Consistencia de Tipos	137
4.5.2 Compatibilidad de Tipos	139
4.5.3 Punteros	141
4.5.4 La Estructura de Tipos de Ada	144
4.6 Tipos Abstractos de Datos	148
4.6.1 El Mecanismo "Class" de SIMULA 67	149
4.6.2 Abstracción y Protección: CLU y Ada	155
4.6.2.1 CLU	155
4.6.2.2 Ada	158
4.7 Modelos de Implementación	162
4.7.1 Tipos Predefinidos y Tipos No Estructurados Definidos por el Usuario	163
4.7.2 Tipos Estructurados	164

	Págs.
4.7.2.1 Producto Cartesiano	164
4.7.2.2 Aplicación Finita	165
4.7.2.3 Secuencias	167
4.7.2.4 Unión Discriminada	168
4.7.2.5 Conjunto Potencia	169
4.7.2.6 Punteros	170
4.7.3 "Clases" y Tipos Abstractos de Datos	171
4.7.4 Recolección de Residuos	174
5. ESTRUCTURAS DE CONTROL	
5.1 Estructuras de Control a Nivel de Sentencia	185
5.1.1 Secuencia	185
5.1.2 Selección	185
5.1.3 Iteración	189
5.1.4 Medida de las Estructuras de Control a Nivel de Sentencia	194
5.1.5 Estructuras de Control Definidas por el Usuario	197
5.1.5.1 Implementación de Modelos para Iteradores	198
5.2 Estructuras de Control a Nivel de Unidad	201
5.2.1 Llamadas Explicitas a Unidades Subordinadas	201
5.2.1.1 Datos como Parámetros	203
5.2.1.2 Subprogramas como Parámetros	207
5.2.2 Llamadas Implicitas a Unidades	210
5.2.2.1 Manejo de Excepciones en PL/I	211
5.2.2.2 Manejo de Excepciones en CLU	213
5.2.2.3 Manejo de Excepciones en Ada	215
5.2.3 Unidades Simétricas: Corrutinas en SIMULA 67	216
5.2.3.1 Modelo de Implementación	218
5.2.4 Unidades Concurrentes	218

	Págs.
5.2.4.1 Semáforos	219
5.2.4.2 Monitores	221
5.2.4.3 Rendezvous	224
5.2.4.4 Análisis Comparativo	226
5.2.4.5 Características de Implementación	227
6. CORRECCION	
6.1 Corrección y Fiabilidad	243
6.2 Revisión de Programas	245
6.2.1 Características Dañinas de los Lenguajes	246
6.2.1.1 Efectos Laterales	246
6.2.1.2 Alias	247
6.2.2 Características de los Lenguajes Disciplinados	250
6.3 Comprobación del Programa	253
6.3.1 Comprobaciones Estáticas	254
6.3.2 Comprobaciones en Tiempo de Ejecución	254
6.3.3 Prueba	254
6.3.4 Ejecución Simbólica	255
6.4 Verificación del Programa	256
6.4.1 Semánticas Formales	256
6.4.2 Verificación de Programas	260
7. PROGRAMACION A GRAN ESCALA	
7.1 ¿Qué es un programa grande?	270
7.2 ¿Programación a Pequeña o a Gran Escala? Métodos de Diseño	272
7.3 Características del Lenguaje para la Programación a Gran Escala	274
7.3.1 Pascal	274
7.3.1.1 Módulos	274

22. Conceptos de Lenguajes de Programación

	Págs.	Págs.	
7.3.1.2 Estructura del Programa	275	8.4.1.1 Objetos	317
7.3.1.3 Independencia Modular	277	8.4.1.2 Funciones	318
7.3.2 SIMULA 67	278	8.4.1.3 Formas Funcionales	320
7.3.2.2 Módulos	278	8.4.1.4 Características no Funcionales de Lisp	320
7.3.2.2 Estructura del Programa	278	8.4.2 APL	321
7.3.2.3 Independencia Modular	280	8.4.2.1 Objetos	321
7.3.3 Ada	280	8.4.2.2 Funciones	321
7.3.3.1 Módulos	280	8.4.2.3 Formas Funcionales	323
7.3.3.2 Estructura del Programa	280	8.4.2.4 Un Programa en APL	326
7.3.3.3 Independencia Modular	283	8.5 Comparación entre Lenguajes Funcionales e Imperativos	328
7.3.4 Un Escenario Ideal	285		
7.4 Herramientas para el Desarrollo de Sistemas	287		
7.4.1 Ejemplo de un Sistema de Desarrollo de Programas	292	9. DISEÑO DEL LENGUAJE	
8. PROGRAMACION FUNCIONAL	301	9.1 Criterios de Diseño	336
8.1 Características de los Lenguajes Imperativos	301	9.1.1 Facilidad de Escritura	337
8.1.1 Un Programa Imperativo	302	9.1.1.1 Simplicidad	337
8.1.2 Problemas de los Lenguajes Imperativos	303	9.1.1.2 Expresividad	339
8.2 La Esencia de la Programación Funcional	306	9.1.1.3 Ortogonalidad	340
8.2.1 Funciones	306	9.1.1.4 Definición	342
8.2.2 Funciones Matemáticas Frente a Funciones de Lenguajes de Programación	308	9.1.2 Legibilidad	343
8.2.3 Lenguajes Funcionales (o Aplicativos)	309	9.1.3 Fiabilidad	346
8.3 Un Lenguaje Funcional Puro y Simple	310	9.2 Implementación del Lenguaje	348
8.3.1 Funciones Primitivas	310		
8.3.2 Formas Funcionales	312		
8.4 Características Funcionales en los Lenguajes existentes	316		
8.4.1 Lisp	317		

GLOSARIO DE LENGUAJES DE PROGRAMACION SELECCIONADOS

	Págs.
Ada	357
ALGOL 60	358
ALGOL 68	359
APL	360
Bliss	361
C	362
CLU	363
COBOL	364
Euclid	365
FORTRAN	366
Gypsy	368
LISP	369
Mesa	370
Modula	371
Pascal	372
Pascal Concurrente	373
PL/I	374
FLZ	375
SIMULA 67	377
SNOBOL 4	378
 BIBLIOGRAFIA	 381

INTRODUCCION

Este libro versa sobre lenguajes de programación. Pero estos lenguajes no existen en el vacío sino que son herramientas para escribir programas. Cualquier estudio sobre lenguajes de programación debe tener en cuenta este hecho. Por tanto, comenzaremos hablando del proceso de desarrollo de los programas y del papel que los lenguajes juegan en este proceso. El propósito de este capítulo es proporcionar una perspectiva desde la cual tener una visión de los lenguajes de programación y de sus posibles usos. Desde este punto de vista, estudiaremos las ventajas de muchos de los conceptos de los lenguajes discutidos en el resto del libro.

1.1 EL PROCESO DE DESARROLLO DE LOS PROGRAMAS

El proceso de desarrollo de un programa se puede dividir en cinco fases o etapas secuenciales. Cada fase puede poner de manifiesto alguna deficiencia en una anterior, en cuyo caso ésta deberá ser repetida. Cada fase involucra una actividad y unos resultados en un conjunto de productos claramente identificables. Estas fases se describen a continuación.

i. Análisis y especificación de necesidades

Un sistema se desarrolla con el fin de satisfacer unas necesidades detectadas por una comunidad de usuarios. Las necesidades de los usuarios se dan como un conjunto de requisitos o especificaciones. Estos requisitos se especifican conjuntamente por los usuarios y por las personas que se dedicarán al desarrollo del programa. El éxito de éste último se medirá finalmente por lo cercano que se encuentre a los requisitos, por la cercanía de éstos a las necesidades expresadas por los usuarios, y por la cercanía de estas últimas a las necesidades reales.

El resultado de esta fase es un documento de especificaciones indicando qué debe hacer el sistema, junto con un manual de usuario, un estudio de viabilidad y coste, eficiencia necesaria, etc. El documento de especificaciones no indicará cómo ha de implementarse el sistema para cumplir estas especificaciones.

ii. Diseño y especificaciones del programa

A partir del documento anterior, el siguiente paso es establecer la arquitectura del sistema. El resultado de esta fase es un documento de especificaciones de diseño que identifica todos los módulos que componen el sistema y sus interconexiones. De las distintas filosofías de diseño y especificaciones de programas existentes actualmente, ninguna es universalmente aceptada. El método de diseño elegido puede tener una

gran influencia en la elección del lenguaje de programación que se utilice en la fase de implementación del sistema. Volveremos a este punto más adelante dentro de este mismo capítulo.

iii. Implementación (codificación)

El sistema se implementa con arreglo al diseño especificado en la fase 2. Este es el único paso en el que se utiliza directamente un lenguaje de programación. El resultado es un sistema totalmente realizado y documentado.

iv. Certificación

El propósito de este paso es asegurar que el sistema cumple las especificaciones establecidas previamente. Normalmente se hace contrastando el sistema con las especificaciones de diseño, dando por hecho que las especificaciones de diseño cumplen perfectamente los requisitos; sin embargo, esto último se debería comprobar antes de que el sistema se pueda dar por certificado.

El resultado del paso 4 es un sistema certificado, que se puede entregar al usuario. Previamente cada uno de los programadores habrá realizado pruebas parciales durante la fase de implementación. Hay tres tipos de pruebas: pruebas a nivel de módulo, pruebas de integración, y una prueba final del sistema. La prueba modular se realiza por cada uno de los programadores que hacen los distintos módulos, asegurándose de que se cumplen las especificaciones de interconexión con el resto del sistema. Las pruebas de integración se realizan mediante la interconexión de algunos módulos con el fin de encontrar inconsistencias entre ellos. La prueba del sistema se realiza durante la fase final de certificación, para asegurar que el funcionamiento global del sistema se ajusta al documento de especificaciones.

Junto a las pruebas, e incluidas también en la fase de certificación, están todas las actividades relacionadas con la evaluación de la corrección del programa; por ejemplo, se realizan en esta fase los intentos de comprobar la verificación formal del sistema.

v. Mantenimiento

Una vez entregado el sistema, pueden existir cambios en él por diversas causas, bien por mal funcionamiento (errores) o debido a que es necesario añadir o mejorar alguna función. Estos cambios constituyen lo que se denomina "mantenimiento del

sistema". La importancia de esta fase puede verse en el hecho de que los costes de mantenimiento son iguales o mayores que los de los pasos anteriores combinados.

A continuación examinaremos el papel que el lenguaje de programación juega en el proceso de desarrollo, mostrando las relaciones entre el lenguaje y otras herramientas de desarrollo de programas (Sección 1.2), así como la relación entre el lenguaje y los métodos de diseño (Sección 1.3).

1.2 EL LENGUAJE COMO UN COMPONENTE MÁS DE UNA UTILIDAD GLOBAL PARA EL DESARROLLO DE PROGRAMAS.

El trabajo en cualquiera de las cinco fases de desarrollo se puede realizar utilizando herramientas de ayuda soportadas por ordenador. La fase mejor soportada es normalmente la de codificación, que utiliza herramientas tales como editores de texto, compiladores, montadores y bibliotecas. Estas herramientas van evolucionando gradualmente conforme se van haciendo evidentes nuevas necesidades de automatización. En los primeros días de la programación, y no sólo en los primeros, se tenía que perforar un programa en tarjetas, tomar las rutinas necesarias de una biblioteca también de tarjetas, combinar todas las rutinas, y enviarlas al compilador. Las posibilidades de error derivadas por ejemplo, del desgaste de las tarjetas de la biblioteca, eran muy grandes. En muchas de las áreas de trabajo en torno a los ordenadores, los avances están dirigidos a la automatización de estas tareas. Ahora para crear un programa, podemos utilizar un editor interactivo y un gestor de archivos para guardarlos en un biblioteca para futuros usos. Cuando sea necesario, varios programas previamente creados y (posiblemente) compilados, se podrán montar para producir un programa ejecutable. Estas herramientas de ayuda incrementan la productividad y reducen las posibilidades de error.

Como hemos visto, el desarrollo de programas supone mucho más que la codificación. Si queremos aumentar la productividad del desarrollo, deberíamos disponer de ayudas mediante ordenador para todas las fases descritas anteriormente.

Entendemos por una utilidad global para el desarrollo de programas, un conjunto integrado por técnicas y herramientas que ayudan al desarrollo de los mismos. La utilidad se usa en todas las fases de dicho desarrollo, es decir, en especificaciones, diseño, implementación, certificación y mantenimiento.

Un escenario ideal para el uso de tales utilidades puede ser el siguiente. Un grupo de especialistas de aplicaciones interactuando con la utilidad desarrollan las especificaciones del sistema. Dicha utilidad lleva control de las especificaciones a medida que se desarrollan y actualizan, y previene contra inconsistencias e incompletitud. También asegura la puesta al día de la documentación, incluyendo los cambios que se hagan a las especificaciones.

Una vez finalizadas las especificaciones, los diseñadores del sistema trabajan sobre otro programa de la utilidad conforme diseñan el sistema: es decir, conforme van especificando los módulos necesarios, así como su interconexión. Las especificaciones de pruebas también se deben realizar en esta etapa. Los implementadores realizan posteriormente su trabajo basándose en el diseño. Las herramientas proporcionadas para la ayuda al desarrollo en esta fase son las más familiares. Incluyen lenguajes de programación, editores, compiladores, simuladores, intérpretes, montadores, y otras que normalmente se usan en la programación.

Es importante que estas herramientas, no sólo funcionen bien, sino que además sean compatibles con las herramientas usadas en otras fases. Por ejemplo, el lenguaje de programación debe ser compatible con algún método de diseño ofrecido por la utilidad global en la fase de diseño. Si se utiliza un método de diseño descendente, el lenguaje debe soportarlo en el sentido de permitir que los niveles jerárquicos de diseño estén presentes en el programa. La siguiente sección está dedicada al influjo que los métodos de diseño ejercen sobre los lenguajes de programación. Lo que queremos recalcar al lector es que las facilidades deben ser vistas como una mezcla de herramientas y métodos, para lo cual es conveniente que sean compatibles entre sí, pudiéndose así sacar un mayor provecho.

En la fase de certificación, la ayuda proporcionada por las utilidades puede ser bastante variada. La utilidad se podría usar para agrupar los datos de prueba generados durante las fases de especificación e implementación. Con lo cual se puede asegurar que las pruebas están permanentemente actualizadas (por ejemplo al hacer un cambio en las especificaciones). También podrían soportar la utilización de especificaciones formales y de verificación de módulos de acuerdo a estas especificaciones. Los resultados de las pruebas podrían ser clasificados y agrupados. Con los resultados de estas pruebas se podría determinar si la estrategia de pruebas seguida es buena o no.

El resultado de toda esta interacción con la utilidad global es un sistema totalmente certificado y documentado, que se corresponde con el conjunto de requisitos establecidos inicialmente. Si en el futuro fuera necesario modificar el sistema, es decir, sus especificaciones, la utilidad debe ser capaz de hacer llegar el cambio a las especificaciones de diseño, al código, a las pruebas de datos y a la documentación, identificando todas las áreas afectadas.

El anterior escenario da una idea simple e idealizada del uso de las ayudas en el desarrollo de programas. Aunque actualmente esta utilidad no existe, nuestra tecnología no está muy lejos de hacerla posible. (De hecho, todos sus componentes existen, y sólo es necesario un trabajo de integración de los mismos en un sistema unificado). En cualquier caso el fin de esta discusión es centrar la atención en las actividades que realizan las personas que se dedican al desarrollo de programas y en el

papel que juegan los lenguajes de programación en este trabajo, en si esas actividades están soportadas o no por un ordenador, y en el hecho de que los lenguajes de programación son sólo una herramienta más de las utilizadas por las personas que se dedican al desarrollo de programas. Como la utilidad de esta herramienta sólo puede ser medida por su contribución al proceso de desarrollo, es necesario examinar las relaciones entre los lenguajes de programación y los otros componentes de la utilidad global.

Hemos de mencionar que el punto de vista expuesto anteriormente, no ha sido siempre aceptado. Ciertamente, los primeros lenguajes de programación estaban diseñados para "programar" en lugar de estarlo para "desarrollar programas". No obstante, si un lenguaje no fue diseñado teniendo en mente la producción de software, debe evaluarse en función del objetivo para el que fué diseñado. Este es el criterio que se ha seguido en este libro.

1.3 METODOS PARA EL DISEÑO DE PROGRAMAS Y LENGUAJES DE PROGRAMACION

Como mencionamos anteriormente, existe una estrecha relación entre los métodos de diseño y los lenguajes de programación. Y este hecho es independiente de si vemos o no al lenguaje como un componente de la utilidad general de desarrollo. Cara a un cierto método de diseño, veremos que unos lenguajes son más adecuados que otros.

Los lenguajes más antiguos, como el FORTRAN, no estaban pensados para soportar un método específico de diseño. Por ejemplo, la ausencia de estructuras adecuadas de control de alto nivel en FORTRAN hace difícil sistematizar el diseño de algoritmos en forma descendente. Contrariamente, el Pascal se diseñó con el fin explícito de soportar el diseño descendente y la programación estructurada. La tendencia cada vez más aceptada es la de que los lenguajes soporten un cierto método de diseño.

Los campos de investigación en los métodos de diseño de programas y en el diseño de lenguajes pueden converger, y en algunos casos ya lo han hecho. El ejemplo más relevante de esta tendencia es la filosofía del ocultamiento de la información (como un método de diseño) y la abstracción de datos (como un principio en el diseño de lenguajes).

El ocultamiento de la información es una técnica de diseño para descomponer sistemas en módulos. Cada módulo esconde un secreto, que puede ser el formato de una estructura particular de datos (registro, fichero, etc). El módulo proporciona funciones de acceso que se pueden utilizar para consultar o actualizar la información contenida en la estructura de datos. Entonces los usuarios del módulo, es decir otros módulos, pueden acceder, llamando a funciones, a la información que el módulo está dispuesto a proporcionar, pero no podrán acceder de forma

directa a la información, manipulando la estructura de datos, o usando variables globales.

Los lenguajes orientados a la abstracción de datos, como SIMULA 67, Mesa, CLU y Ada, son capaces de soportar esta filosofía. Una "class" de SIMULA, un "module" de Mesa, un "cluster" de CLU, o un "package" de Ada pueden representar directamente un módulo con información oculta.

No está claro si el concepto del ocultamiento de la información coincide con la visión que tiene SIMULA de la "class" como un mecanismo para describir abstracciones. Es probable que los conceptos de ocultamiento de la información y "class" de SIMULA hayan progresado separadamente y se hayan encontrado en su madurez. La conclusión es que la elección del método de diseño del programa puede influir en el diseño de un lenguaje, así como en la elección del lenguaje a utilizar.

1.4 LA ARQUITECTURA DEL ORDENADOR Y LOS LENGUAJES DE PROGRAMACIÓN

Los métodos de diseño influyen en los lenguajes en el sentido de establecer condiciones que el lenguaje debe cumplir. La arquitectura del ordenador ejerce una influencia en el sentido contrario, es decir tiende a limitar el lenguaje para que éste se pueda implementar eficientemente en las máquinas actuales. De hecho, los lenguajes han estado restringidos por las ideas de Von Neumann, ya que los ordenadores más usados tienen una estructura muy similar a la original de Von Neumann. Y lo que es aún peor es que la mayoría de las arquitecturas dirigidas a lenguajes de alto nivel no están en realidad orientadas a las necesidades reales del lenguaje, ya que en un principio, el lenguaje que motivó la arquitectura estaba a su vez influenciado por una arquitectura particular de un ordenador concreto.

Backus (Backus 1978) ha combatido activamente este punto de vista, animando al uso de un estilo funcional de programación por dos causas fundamentales: por su más simple (y más robusto) fundamento matemático y por las altas prestaciones que proporciona. LISP es el ejemplo más conocido de un lenguaje que siga este estilo, pero muchas de las características funcionales que en un principio tenía el LISP se han modificado para poder tener una implementación eficiente del lenguaje en las arquitecturas de las máquinas tradicionales. Backus dice que sólo abandonaremos los lenguajes tradicionales y adoptaremos lenguajes radicalmente distintos cuando existan ordenadores que soporten eficientemente estos nuevos lenguajes. También predice que la tecnología del hardware hará pronto esto posible.

Backus critica el concepto de variables y de sentencias de asignación, diciendo que son la raíz de muchos males y una maldición transmitida por la arquitectura de Von Neumann. Sin embargo se observa que a pesar de los años transcurridos, los lenguajes de programación funcional, como el LISP, sólo han

tenido éxito a la hora de reemplazar a los lenguajes convencionales, en ciertas áreas de aplicación. Además de los problemas de implementación, existen probablemente cuestiones sicológicas y sociológicas que dificultan el alejamiento de los lenguajes tradicionales. En la actualidad, estos problemas son objeto de controversia, y el debate sobre la programación funcional contra la tradicional se encuentra actualmente en una etapa de investigación.

Este libro considera los lenguajes que han prevalecido durante los últimos 25 años y las cuestiones que influyeron en su diseño. Nuestro estudio se ha centrado en el examen de los denominados lenguajes orientados a sentencias, sin embargo, en el Capítulo 8 se estudian los aspectos más importantes de los lenguajes de programación funcional.

1.5 OBJETIVOS EN EL DISEÑO DE LENGUAJES IMPUESTOS POR EL PROCESO DE DESARROLLO DE LOS PROGRAMAS.

Volviendo al tema de ver un lenguaje de programación como una herramienta para el desarrollo de programas, esta sección señala los requisitos que desde este punto de vista han de cumplir los lenguajes de programación.

i. Los programas deben ser fiables

En otras palabras, los usuarios deben confiar en los programas. Deben sentirse a gusto usándolos, incluso en la presencia de casos no frecuentes e indeseables, como pudieran ser los fallos del hardware. Esta propiedad informal y difícil de cuantificar está fuertemente ligada con otra propiedad más formal que es la corrección. Un programa es correcto si se comporta de acuerdo a sus especificaciones: cuanto más rígidas y menos ambiguas sean las especificaciones, más convincentemente se puede probar la corrección del programa. La fiabilidad es un requisito que ha ido ganando importancia a medida que ha ido aumentando la complejidad de las tareas asignadas a los programas.

ii. Los programas deben ser mantenibles

Conforme el coste del software sube y se incrementa la complejidad de los sistemas que tienen que desarrollarse, consideraciones económicas tienden a reducir la posibilidad de desechar programas ya existentes para desarrollar aplicaciones similares. Los programas existentes deben poder ser modificados para aceptar nuevas necesidades. Además, como para muchos sistemas complejos es prácticamente imposible obtener inicialmente los requisitos reales, el método seguido muchas veces consiste en hacer evolucionar el sistema aproximándolo gradualmente hacia el deseado.

iii. Los programas se deben ejecutar eficientemente

La eficiencia ha sido siempre un objetivo en todo sistema de programación. Este objetivo afecta tanto al lenguaje utilizado (sus características han de poder ser implementadas eficazmente en las arquitecturas actuales) como a la elección de los algoritmos a utilizar.

Estos tres objetivos, fiabilidad, mantenibilidad y eficiencia, pueden cumplirse utilizando herramientas apropiadas en el desarrollo de los programas. A continuación se tratan aquellos aspectos del lenguaje que influyen directamente en la consecución de los tres objetivos citados.

1.5.1 Lenguaje y Fiabilidad

Las siguientes cualidades de los lenguajes de programación contribuyen al objetivo de la fiabilidad de los programas.

Facilidad de escritura. Esta es una propiedad difícil de cuantificar. Básicamente se refiere a la posibilidad de expresar un programa en una forma que sea natural para el problema. El programador no debe distraerse con detalles y trucos del lenguaje, sino que se debe dedicar a la actividad más importante, que es solucionar el problema. Aun sabiendo que es un criterio subjetivo, normalmente se está de acuerdo en que es más fácil escribir en un lenguaje de alto nivel que en uno que no lo sea (ensamblador, lenguaje máquina). Por ejemplo, un programador de ensamblador se distrae a menudo con el mecanismo de direccionamiento necesario para acceder a determinados datos, posicionando registros, índices, etc. Cuanto más fácil sea concentrarse en la resolución del problema, y no en la escritura del programa, menos propensa al error será esta última.

Legibilidad. Ha de ser posible seguir la lógica del programa y descubrir la posible presencia de errores, mediante un examen del mismo. La legibilidad es también un criterio subjetivo que depende en gran medida de gustos y estilos. No obstante, cuanto más simple sea el lenguaje y más naturalmente permita expresar los algoritmos, más sencillo será entender lo que hace un programa examinando su código. Por ejemplo, la sentencia goto puede hacer muy difícil la lectura de un programa, ya que puede hacer imposible leerlo de principio a fin y entender su funcionamiento. En lugar de ello, uno se puede ver obligado a avanzar y retroceder a la búsqueda del lugar donde nos envió la última sentencia goto. A pesar de que muchos programadores no se ponen de acuerdo en si la sentencia goto es perniciosa o no, en lo que sí parece existir un consenso general, es que en la mayoría de los casos la sentencia goto no debe ser usada.

Como veremos, muchas de las cualidades que hacen que un

programa sea legible por las personas, le hacen también más fácilmente procesable por un ordenador. La comprobación automática realizada por ordenador contribuye muy efectivamente a la corrección de los programas.

Posibilidad de tratar excepciones. El lenguaje debe poder permitir el tratamiento de posibles sucesos no deseados (desbordamientos aritméticos, entradas inválidas, etc.) y la posibilidad de dar respuestas adecuadas a estos acontecimientos. De esta manera, el comportamiento del sistema llega a ser totalmente predecible, incluso en presencia de situaciones anómalas.

1.5.2 Lenguaje y Mantenibilidad

La necesidad de mantener un programa impone dos requisitos a los lenguajes de programación: los programas escritos en un lenguaje deben ser legibles y deben ser modificables. De igual forma que la legibilidad (discutida anteriormente), la modificabilidad de un programa es algo subjetivo. Sin embargo, es posible hablar de características que hacen que un programa sea más modificable. Por ejemplo, hay lenguajes que permiten dar nombres simbólicos a las constantes. La elección de un nombre apropiado para una constante mejora la legibilidad del programa (por ej., la utilización de pi en lugar de 3.1416). Así, si en un futuro se ha de cambiar el valor de la constante, sólo tendremos que modificar la definición de la constante, en lugar de tener que cambiarla en todos los sitios donde se use.

Ejemplo (Pascal)

```
const numero_de_palabras = 65536;
var memoria: array[1..numero_de_palabras] of integer;
```

```
    . . .
    . . .
    . . .
if m > numero_de_palabras then . . . fin_de_memoria . . .
```

La constante simbólica numero_de_palabras representa el tamaño de la memoria. Cada elemento de la memoria es un entero al que se puede acceder por medio de un subíndice, cuyo rango es de 1 a numero_de_palabras. Si en algún momento la memoria se aumenta, todo lo que necesitaríamos sería cambiar la primera línea del programa, aunque esta información se utilice en otros sitios del mismo.

1.5.3 Lenguaje y Eficiencia

La necesidad de eficiencia ha guiado desde siempre el diseño de los lenguajes. Es más, muchos de los lenguajes han considerado a la eficiencia, de una forma implícita o explícita, como el objetivo fundamental de su diseño. Por ejemplo, el FORTRAN se diseñó en un principio para una máquina específica (el IBM 704). Muchas de las restricciones del FORTRAN, tales como el número de dimensiones de una matriz, o la forma de las expresiones utilizadas como índices de una matriz, se basan directamente en lo que se podía implementar eficientemente en el IBM 704.

Sin embargo, el problema de la eficiencia ha cambiado notablemente. La eficiencia, ya no sólo se mide por la velocidad de ejecución y por el espacio ocupado por el programa. El esfuerzo inicial de desarrollo y el necesario para su mantenimiento posterior también han de ser tenidos en cuenta a la hora de medir la eficiencia. Y en ello, una vez más, el lenguaje de programación juega un importante papel.

Un lenguaje es una herramienta eficiente si tiene las cualidades de ser fácil de escribir, ser mantenible y además ser "optimizable". La facilidad de escritura y la mantenibilidad ya se han discutido anteriormente. La "optimización" se refiere a la cualidad de permitir una optimización automática del programa.

La optimización es una característica importante, ya que gran parte del tiempo empleado tradicionalmente en programación, se dedica a encontrar formas eficientes de hacer las cosas. Sin embargo, la preocupación por la optimización se debería eliminar de las primeras etapas de la programación. El camino ideal debería ser, en primer lugar producir un programa del que se pueda demostrar su corrección, y después, a través de una serie de transformaciones, modificarlo para obtener otro que sea correcto y eficiente. Un lenguaje es optimizable, si hace posible la aplicación automática de estas transformaciones. Por ejemplo la existencia de sentencias *goto* complica la optimización automática. En general, podemos decir que muchas de las características que reducen la optimización, impiden también la legibilidad de un programa.

1.6 UNA BREVE PERSPECTIVA HISTÓRICA

Esta sección examina brevemente los desarrollos en el diseño de lenguajes, siguiendo la evolución de las ideas y los conceptos desde una perspectiva histórica.

El proceso del desarrollo de programas consistía en un principio únicamente en la fase de codificación. En aquellos lejanos días el ordenador sólo se utilizaba en aplicaciones científicas. Una aplicación la desarrollaba una sola persona. El problema a resolver (por ejemplo, una ecuación diferencial) era muy preciso y bien entendido. Como resultado de ello, no había mucha necesidad de un análisis de requisitos, de unas

especificaciones de diseño, o incluso de mantenimiento. Por tanto, un lenguaje de programación sólo tenía que soportar a un programador que programaba lo que hoy consideraríamos una aplicación muy sencilla.

El deseo de aplicar el ordenador en más y más aplicaciones, hizo que se comenzase a utilizar en entornos más sofisticados y menos entendibles. Esto a su vez condujo a la necesidad de "agrupar" a los programadores y a hacer más formal el acceso al ordenador. Las fases de especificaciones y diseño que hasta entonces las hacía un solo programador, ahora requerían un grupo, y los resultados debían ser comunicados de una persona a otra. Y ya que tanto esfuerzo y dinero se gastaba en el desarrollo, los viejos sistemas no debían desecharse cuando se necesitase un nuevo. Consideraciones económicas persuadieron de las ventajas que suponía mejorar un sistema existente para cubrir nuevas necesidades. De esta forma, el mantenimiento de un programa se transformó también en una cuestión a tener en cuenta.

La fiabilidad de un sistema es otro problema que ha ido ganando importancia gradualmente, debido a dos factores fundamentales. Uno de ellos es que en general los sistemas se hacen para usuarios con una formación pequeña o nula en ordenadores, y estos usuarios no son tan tolerantes a los fallos del sistema como lo son los que lo han desarrollado. El segundo factor, es que los sistemas se utilizan ahora en áreas como centrales nucleares y monitorización de pacientes, donde cualquier fallo puede tener consecuencias desastrosas.

Las deficiencias de los lenguajes en estas áreas, ha orientado una gran cantidad de esfuerzo en el campo del diseño de nuevos lenguajes. El propósito de este libro es examinar estas influencias sobre el diseño de los lenguajes y ver hasta qué punto han alcanzado su objetivo. La tabla I da una genealogía de la mayoría de los lenguajes tratados en este libro.

TABLA I. Genealogía de los Lenguajes de Programación

Lenguaje	Año	Creador	Lenguaje predecesor	Propósito
FORTRAN	1954-57b	J. Backus (IBM)	-----	Cálculo numérico
ALGOL 60	1958-60c	Comité	FORTRAN	Cálculo numérico
COBOL	1959-60c	Comité	-----	Gestión
APL	1956-60b	R. Iverson (Harvard)	-----	Tratamiento de matrices
LISP	1956-62b	J. McCarthy (MIT)	-----	Cálculo simbólico
SNOBOL	1962-66b	R. Griswold (Labs. Bell)	-----	Trat. de caracteres
PL/I	1963-64c	Comité IBM	FORTRAN ALGOL 60 COBOL	Propósito general
SIMULA 67	1967a	O. J. Dahl y Otros (Centro de Cálculo Noruego)	ALGOL 60	Propósito general y simulación
ALGOL 68	1968-68c	Comité	ALGOL 60	Propósito general
Bliss	1971a	Wulf y Otros (Universidad de Carnegie Mellon)	ALGOL 68	Prog. de Sistemas
Pascal	1971a	N. Wirth (ETH Zurich)	ALGOL 60	Propósito general. Educación. Soporte de la programación estructurada.
C	1974a	D. Richie (Labs. Bell)	ALGOL 68 BCPL	Prog. Sistemas

Lenguaje	Año	Creador	Lenguaje predecesor	Propósito
Mesa	1974c	Xerox PARC	Pascal SIMULA 67	Prog. Sistemas
Pascal	1975a	P. Brinch Hansen (Cal. Tech)	Pascal	Prog. Concurrente
Concurrente				
CLU	1974-77b	B. Liskov y Otros (MIT)	SIMULA 67	Soporte de métodos basados en la abstracción.
Euclid	1977a	Comité	Pascal	Verificación de programas.
Gypsy	1977a	Good y Otros (Universidad Texas-Austin)	Pascal	Verificación de programas.
PLZ	1977a	Zilog Inc.	Pascal	Prog. Sistemas
Modula	1977a	N. Wirth (ETH Zurich)	Pascal	Prog. Sistemas. Tiempo real
Ada	1979a	J. Ichbiah y otros. (CII Honeywell Bull)	Pascal SIMULA 67	Propósito general. Sistemas empotrados. Tiempo real.

a- Primera descripción oficial del lenguaje.

b- Implementación inicial y diseño del lenguaje.

c- Diseño del Lenguaje.

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

En los últimos años se han publicado numerosos textos que discuten el ciclo de vida del desarrollo de un programa. (Tausworthe 1977) da una visión de todos los aspectos de la producción de programas y hace una propuesta para normalizar su desarrollo. Otros textos sobre la producción de software son (Aron 1974), (Yourdon 1975), (Jackson 1975), (Myers 1976), (Zelkowitz y otros 1979).

La fase de programación se discute en profundidad en una publicación especial del ACM Computing Surveys (ACM-CS 1974). Se hace especial énfasis en el estilo de la programación y en la generación sistemática de programas. En particular el artículo de Knuth discute el uso y el abuso de los goto's, y en el artículo de Wirth se describe la forma de escribir programas correctamente estructurados en Pascal. Este lenguaje es adecuado para soportar métodos de diseño descendente, la programación por refinamientos sucesivos o paso a paso (Wirth 1971b). El origen de muchas ideas sobre la programación sistemática puede estar en Dijkstra (1968a). Este artículo es el punto de partida de una amplia investigación en "programación estructurada" a comienzos de la década de los 70. (Dahl y otros, 1972) es una excelente referencia sobre dicho tema. Un punto de vista más elemental se tiene en la introducción del libro de texto para la programación de N. Wirth (Wirth 1973).

El principio del "ocultamiento de la información" lo propuso (Parnas 1972b) como la base para un método de diseño que soporte modularidad, legibilidad y modificabilidad de los programas. Referencias adicionales a este punto son (Myers 1975), (Myers 1976) y (Myers 1978).

Se va haciendo cada vez más popular la idea de un entorno de programación que soporte el desarrollo de programas con una gran cantidad de herramientas. (Kernighan y Mashey 1979) presentan una perspectiva general de UNIX. Otro sistema de desarrollo de programas es el descrito en (Cheatham y otros 1979). (Sandewall 1978) discute un entorno para LISP. Los requisitos para el entorno de programación de los programas escritos en Ada se especifican en (DOD 1980a).

(Wegner 1976) y (ACM-SIGPLAN 1978) proporcionan una perspectiva histórica de los desarrollos de los lenguajes de programación. (Backus 1978a) argumenta que tales desarrollos presentan una evolución no real y defiende un estilo de programación puramente funcional.

EVOLUCION DE CONCEPTOS EN LOS LENGUAJES DE PROGRAMACION

Creo que ahora estamos por fin a punto de descubrir cómo deberían ser realmente los lenguajes de programación. Espero con ilusión ver muchos experimentos serios con el diseño de lenguajes durante los próximos años; y mi sueño es que en 1984 lleguemos a un acuerdo para desarrollar un lenguaje de programación realmente bueno (o más precisamente una familia coherente de lenguajes). Además, pienso que la gente llegará a estar tan desencantada de los lenguajes que están usando ahora (incluidos COBOL y FORTRAN) que este nuevo lenguaje, UTOPIA 84, tendrá una gran probabilidad de triunfar. Ahora mismo estamos lejos del objetivo, sin embargo hay ya indicios para pensar que tal lenguaje está tomando forma poco a poco... Se debe realizar una gran cantidad de investigación si queremos tener el deseado lenguaje en 1984. (Knuth 1974).

Debemos reconocer la innegable y poderosa influencia que nuestro lenguaje ejerce sobre nuestra manera de pensar, y de hecho, define y delimita el espacio abstracto en el que podemos formular y dar forma a nuestros pensamientos. (Wirth 1974).

El lenguaje es el vehículo en el que expresamos nuestros pensamientos, y la relación entre esos pensamientos y nuestro lenguaje es sutil y complicada. La naturaleza del lenguaje realmente conforma y modela la manera en que pensamos... Si, proporcionando las estructuras adecuadas a un lenguaje, podemos mejorar los programas escritos utilizando las mismas, todo el entorno saldrá beneficiado... El diseño de un lenguaje debería proporcionar al menos las facilidades que permitan una expresión comprensible de los algoritmos, y en el mejor de los casos, un lenguaje debería sugerir mejores formas de expresión. Pero un lenguaje no es una panacea. Un lenguaje no puede, por ejemplo, prevenir la creación de programas oscuros; un programador ingenioso siempre puede encontrar un número infinito de vías de ofuscación. (Wulf 1977).

El Capítulo 1 discute las múltiples facetas de la relación entre los lenguajes de programación y el ciclo completo de la producción de software. Por un lado, el lenguaje puede favorecer la adopción de métodos sistemáticos para el diseño de programas; y por otro, el lenguaje tiene una enorme influencia en la fiabilidad, legibilidad y modificabilidad de los programas. Las reglas y métodos que han sido útiles para el desarrollo de programas fiables y de alta calidad pueden ser, y son cada vez más, incorporadas a los lenguajes de programación con el fin de animar a los usuarios a aplicar esos métodos, o al menos, a hacer más fácil su aplicación.

La anterior cita de Knuth expresa una visión quizás optimista del estado actual del arte en el área de los lenguajes

de programación. El propósito de este libro es presentar las ideas más importantes que han influido en la evolución de los diseños de los lenguajes, con el fin de mostrar cómo UTOPIA 84 va poco a poco tomando forma. Este capítulo desarrolla los criterios que se usarán en el libro para clasificar los conceptos de los lenguajes de programación y explicar su evolución.

El capítulo está organizado de la siguiente forma: La Sección 2.1 presenta el concepto de abstracción y muestra su papel fundamental en la programación. En las Secciones 2.2 y 2.3 se analizan dos tipos particulares del concepto anterior, abstracciones de datos y abstracciones de control. La corrección, otro concepto que ha influido en la evolución de los lenguajes de programación, se ilustra en la Sección 2.4. Finalmente, la Sección 2.5 muestra la influencia que ejerce sobre los lenguajes la necesidad de producir programas grandes.

El objetivo de este capítulo es fundamentalmente la presentación. En lugar de examinar las características de los lenguajes en detalle, presentaremos los conceptos y los criterios que serán utilizados a lo largo del libro en la presentación y evaluación de los diferentes lenguajes.

2.1 EL PAPEL DE LA ABSTRACCION

Los ordenadores están reemplazando progresivamente a los hombres en muchas áreas, desde la administración de negocios hasta el control de procesos. Para reemplazar un procedimiento manual, los diseñadores de software deben reproducir su comportamiento en un programa de ordenador. Por lo tanto, los programas de ordenador se pueden ver como modelos de esos procedimientos manuales.

Como cualquier modelo, un programa de ordenador es una abstracción de la realidad. La abstracción es el proceso de identificación de las cualidades o propiedades importantes de los fenómenos que se están modelando. Usando el modelo abstracto, somos capaces de concentrarnos únicamente en las cualidades relevantes o propiedades de los fenómenos, e ignorar las irrelevantes. El qué es relevante depende del propósito para el cual se diseña la abstracción. Por ejemplo, una persona que está aprendiendo a conducir, podría representar un coche simplemente por cuatro entidades: los pedales de acelerador, freno y embrague, y el volante. El ingeniero diseñador del coche debería usar un modelo que también mostrara las relaciones entre los pedales y el motor. Para la abstracción del conductor, el motor es una entidad irrelevante; para el ingeniero, es crucial.

La abstracción es el concepto clave en la teoría de la programación. En concreto, tiene una relación doble con los lenguajes de programación. Por un lado, los lenguajes son las herramientas con las que los diseñadores pueden implementar los modelos abstractos. Por otro, son ellos mismos abstracciones del procesador sobre el cual se implementa el modelo.

Sin embargo, los primeros lenguajes no reconocían el papel crucial que la abstracción juega en la programación. Por ejemplo, a comienzos de la década de los 50, el único mecanismo de abstracción proporcionado por los lenguajes ensambladores sobre los lenguajes máquina era la denominación simbólica. El programador podía usar términos relativamente autoexplicativos para denominar códigos de operación y podía referirse simbólicamente a posiciones de memoria. Así el programador podía abstraerse de la representación de los programas en la máquina, y en particular de la asignación de posiciones de memoria y de la representación en bits de los códigos de operación.

La contribución que esta facilidad supuso para la legibilidad y la modificabilidad de los programas es bien conocida. La gran cantidad de "contabilidad" requerida en la programación en lenguaje máquina es automáticamente realizada por el traductor, y ello hace que la programación sea más fácil y menos propensa al error. Por ejemplo, en un programa de modelos geométricos, el usuario puede usar libremente nombres autoexplicativos, como VOLUMEN, AREA, PERIMETRO, etc. para designar los valores de entidades geométricas, sin necesidad de preocuparse de las posiciones de memoria donde dichos valores están almacenados. Además, en tiempo de traducción se pueden realizar algunos cheques simples de corrección. Por ejemplo, símbolos no definidos o multi-definidos pueden ser detectados por el ensamblador, y este hecho puede ayudar al programador a la producción de programas correctos. Otras herramientas simples, como tablas de referencias cruzadas, pueden proporcionar una ayuda adicional. En nuestro ejemplo, un error originado en la evaluación del volumen de un objeto, se puede descubrir examinando la lista de sentencias que asignan valores a la variable VOLUMEN. Una lista de este tipo la puede proporcionar la tabla de referencias cruzadas.

Los subprogramas (y las macros) fueron también introducidos en los lenguajes tipo ensamblador como un mecanismo del que se servía el programador para denominar una actividad descrita por un grupo de acciones y consideraría como una única acción. En nuestro ejemplo, reducciones de escala, vistas en perspectiva, proyecciones planarias, etc. podrían ser ejemplos de subprogramas útiles.

Los subprogramas son herramientas útiles para la programación metódica, ya que son mecanismos para la construcción de abstracciones. Un subprograma es la implementación de una abstracción, mientras que la llamada al subprograma representa el uso de la abstracción. Al diseñar un subprograma, el programador se concentra en cómo trabaja dicho subprograma. Más tarde, en la llamada al mismo, el programador puede ignorar el "cómo", y concentrarse en qué es lo que hace. Este es otro ejemplo del uso de la abstracción en programación. El subprograma puede verse como una extensión del lenguaje en una nueva operación, pues cuando usa la operación, el programador se abstrae de su implementación real. La máquina se puede ver como un procesador abstracto de propósito especial cuyo repertorio de instrucciones

contiene la nueva operación.

A finales de los 50 y principios de los 60 aparecieron los primeros lenguajes de más alto nivel, los cuales proporcionaron un rico conjunto de mecanismos para definir abstracciones. Estos mecanismos se pueden utilizar para definir abstracciones de datos y abstracciones de control. Las abstracciones de datos modelan los datos manipulados por los programas. Las abstracciones de control modelan las operaciones sobre esos datos, combinando acciones elementales en modelos de complejidad arbitraria. Las Secciones 2.2 y 2.3 explican la evolución de las abstracciones de datos y de control en los lenguajes de programación.

2.2 LAS ABSTRACCIONES DE DATOS

2.2.1 Abstracción de Datos en los Lenguajes Primitivos

Los lenguajes a nivel de la máquina veían los datos almacenados como cadenas de bits que podían ser manipulados por las instrucciones de la máquina. El repertorio de instrucciones incluía operaciones de desplazamiento, lógicas, de aritmética en coma fija, etc.

FORTRAN, COBOL y ALGOL 60 dieron un primer paso hacia la introducción de abstracciones en los datos. En estos lenguajes, la información almacenada en determinadas posiciones de memoria no se ve como una secuencia de bits anónimos, sino como un valor entero, real, lógico, etc. La decisión sobre las abstracciones de datos particulares a incluir en un lenguaje estaba influida fundamentalmente por el tipo de máquinas para las que se diseñaba el lenguaje (por ejemplo, si la máquina proporcionaba aritmética en coma fija o en coma flotante), y por el espectro de aplicaciones que se suponía que los lenguajes iban a cubrir (por ejemplo, si el lenguaje se diseñaba para cubrir aplicaciones científicas).

Como resultado, ningún lenguaje resultó ser apropiado para todas las aplicaciones, ya que el programador estaba limitado por la rigidez del conjunto fijo de abstracciones proporcionado por el lenguaje. Por ejemplo, FORTRAN no es el lenguaje apropiado para problemas de manipulación de cadenas, ni COBOL lo es para resolver un sistema de ecuaciones diferenciales, y ninguno de los dos es aconsejable para manipulación de matrices o para aplicaciones que requieren el manejo de estructuras dinámicas accesibles desde diferentes puntos.

PL/I trató de resolver estos problemas recogiendo muchas de las abstracciones proporcionadas por los lenguajes anteriores, principalmente FORTRAN, ALGOL 60 y COBOL. Sin embargo, muchas de las características del lenguaje orientadas al manejo de datos resultaron difíciles de usar, y a menudo inseguras. En el Capítulo 4 veremos algunos ejemplos. Además, la proliferación de abstracciones incorporadas, en oposición a los mecanismos de definición de nuevas abstracciones, tuvo como principal efecto el

conseguir un lenguaje "grande"; y que sin embargo no cubría todas las necesidades que podían aparecer en diferentes aplicaciones.

2.2.2 Abstracción de Datos en ALGOL 68, Pascal y SIMULA 67

El objetivo perseguido por los lenguajes de la siguiente generación, SIMULA 67, ALGOL 68, Pascal, es de alguna forma menos ambicioso, pero resultó ser más efectivo. Estos lenguajes tratan de alcanzar una generalidad proporcionando no un conjunto exhaustivo de abstracciones incorporadas, sino mecanismos flexibles y fáciles de usar por medio de los cuales el programador puede definir nuevas abstracciones. Este objetivo encaja perfectamente en lo que debe ser un método de diseño basado en el reconocimiento de abstracciones. Además, las abstracciones definidas en la fase de diseño pueden considerarse, hasta cierto punto, como la estructura básica del programa. Como consecuencia, los programas son más fácilmente comprensibles y tienen una mayor probabilidad de ser correctos.

ALGOL 68 y Pascal permiten al programador usar tipos de datos incorporados y constructores de nuevos tipos (matrices, registros, etc.). Por ejemplo, usando la notación de Pascal, las siguientes definiciones y declaraciones definen dos nuevos tipos (estudiante y curso), y tres variables (grupo_A_tarde, grupo_C_mañana y grupo_B_nocturno) del tipo curso.

```
type estudiante = record nombre: array [1..20] of char;
                           apellido_1: array [1..25] of char;
                           apellido_2: array [1..25] of char
end;
curso = record no_de_estudiantes: 0..20;
           asistentes: array [1..20] of estudiantes
end;
```

```
var grupo_A_tarde, grupo_C_mañana, grupo_B_nocturno: curso;
```

El tipo estudiante que acabamos de definir es una estructura de datos con tres componentes, utilizada para guardar una identificación de un estudiante. El tipo curso se define como una tabla de estudiantes (asistentes), junto con el número de estudiantes (no_de_estudiantes: un entero entre 0 y 20) que se matricularon en el curso. Así, cualquiera de las tres variables (por ejemplo, grupo_A_tarde) se puede ver como muestra la Figura 2.1.

no_de_estudiantes	asistentes
16	Gonzalo Abad Valladolid 1
	Angel Diaz Peligro 2

	Cristina Sanchez Rubio 16

	20

Figura 2.1 Estructura de datos de la variable grupo_A_tarde.

Una pregunta inmediata es: ?por qué hemos elegido una representación como la anterior para representar estudiantes y cursos? Y lo que es más, ?por qué necesitamos una estructura de datos para los estudiantes y los cursos? Las respuestas a estas preguntas radican en las especificaciones del programa y en las abstracciones que se decidieron en la etapa de diseño, y es imposible encontrarlas mirando exclusivamente el programa. Sin embargo, sería muy útil disponer de un lenguaje que permita la clarificación de estas cuestiones como parte del propio programa.

Por ejemplo, podemos imaginar que el programa resuelve una aplicación para un profesor que da clase a grupo_A_tarde, grupo_C_mañana y grupo_B_nocturno. Este profesor recibe de la Secretaría del centro un conjunto de fichas que contienen los datos de los alumnos que se han matriculado en dichos cursos, y quiere producir un listado ordenado de los mismos agrupados por cursos.

También podemos imaginar que el tipo curso ha sido creado porque es necesario operar sobre objetos, como grupo_A_tarde, insertando un nuevo estudiante (en el orden apropiado) e imprimiendo los nombres de los alumnos matriculados. Una versión inicial abstracta del programa podría ser:

```
for cada conjunto de fichas do
  sea E la identificación del estudiante y sea C el nombre del
  curso;
  insertar E en la tabla del curso C en el orden apropiado
end-of-do;
```

```
imprimir la tabla de grupo_A_tarde;
imprimir la tabla de grupo_C_mañana;
imprimir la tabla de grupo_B_nocturno;
```

El objetivo (imaginario) perseguido por el diseñador consiste en la caracterización de un conjunto de operaciones lógicas sobre las tablas de los cursos antes de decidir cómo implementar tales tablas en el ordenador. Cualquier forma concreta de implementación de tablas debe proveer una representación de las tablas así como un conjunto de operaciones concretas, es decir, algoritmos expresados en el lenguaje de programación correspondientes a las operaciones lógicas sobre las tablas de los cursos. Sin embargo, la decisión de cómo representar las tablas de los cursos y de cómo implementar las operaciones está a un nivel más bajo de abstracción que la decisión de que las tablas de los cursos son necesarias para almacenar la identificación de los alumnos.

En Pascal (e igualmente en ALGOL 68), esas decisiones conducen a la declaración de los nuevos tipos previamente descritos, y a la implementación de los siguientes procedimientos, uno para cada operación lógica.

- Insertar: recoge un parámetro del tipo curso y otro parámetro del tipo estudiante e inserta el valor del estudiante en el curso.
- Imprimir: recoge un parámetro del tipo curso e imprime su contenido.

Los procedimientos insertar e imprimir están fuertemente relacionados con el tipo curso. Son concretamente las operaciones que manipulan los objetos de tipo curso. Sin embargo, esta relación lógica no aparece tan evidente en un programa escrito en Pascal o en ALGOL 68.

Por otra parte, SIMULA 67 proporciona una estructura (class) que permite que la representación y las operaciones concretas puedan especificarse en una única unidad sintáctica. Esta estructura mejora considerablemente la legibilidad de los programas ya que agrupa las entidades que están envueltas en la implementación de una cierta abstracción.

2.2.3 Hacia los Tipos Abstractos de Datos.

Hay semejanzas entre los tipos definidos por el usuario de Pascal, ALGOL 68 y SIMULA 67, y los tipos predefinidos de esos lenguajes. Por ejemplo, consideremos el tipo predefinido integer y el tipo curso definido por el usuario que hemos visto en la Sección 2.2.2. Los dos tipos son abstracciones construidas sobre una representación interna, una cadena de bits para integer y un registro ("record") para curso. Los dos tipos tienen asociado un conjunto de operaciones, operaciones aritméticas y comparaciones para integer, e insertar e imprimir para curso.

Sin embargo, los tipos predefinidos y los tipos definidos por el usuario difieren uno del otro en un aspecto importante. Los tipos predefinidos ocultan al programador la representación

interna; ésta no puede manipularse directamente. Por ejemplo, el programador no puede acceder a un bit particular de la cadena de bits que representan un integer. Por otra parte, los procedimientos insertar e imprimir no son los únicos medios de manipular un curso. El programador puede operar directamente sobre los componentes de los objetos de tipo curso y no está obligado a utilizar exclusivamente las operaciones definidas para el nuevo tipo. Por ejemplo,

```
grupo_B_nocturno.no_de_estudiantes := 17
```

sería una operación legal, aunque no recomendable, que modifica el número de estudiantes matriculados en el grupo_B_nocturno. En otras palabras, desde el punto de vista del lenguaje, no hay distinción entre dos niveles de abstracción: el nivel en el que se pueden usar cursos como objetos nuevos, y el nivel en el que se puede conocer la representación de los cursos en términos de abstracciones de más bajo nivel. Al mismo tiempo, el programador puede ver los cursos como objetos abstractos manipulables a través de las operaciones insertar e imprimir, y como estructuras de datos concretas cuyos componentes pueden ser accedidos y modificados individualmente.

Esta confusión entre niveles de abstracción puede conducir a la producción de programas difíciles de leer. Y lo que es más importante, reduce la modificabilidad de los mismos. Por ejemplo, supongamos que decidimos cambiar la representación de las tablas a una estructura de lista secuencial o a un árbol binario. El cambio no está localizado dentro de las declaraciones de los datos y de sus operaciones concretas. Es también necesario comprobar todos los accesos directos a la representación de los datos, y estos accesos pueden estar diseminados a lo largo de todo el programa.

En conclusión, si queremos definir nuevos tipos de datos en un programa, sería deseable tener un lenguaje de unas características tales que permitiera (a) la asociación de una representación con sus operaciones concretas en una unidad adecuada del lenguaje que incorpore los nuevos tipos, y (b) el ocultamiento de la representación del nuevo tipo a las unidades que lo usan.

Los tipos definidos por el usuario que satisfacen las propiedades (a) y (b) se denominan tipos abstractos de datos. La propiedad (a) hace que la versión final del programa refleje las abstracciones descubiertas durante la fase de diseño del programa. La estructura resultante del programa llega a ser autoexplicativa. La propiedad (b) enfatiza la distinción entre los niveles de abstracción y favorece la modificabilidad de los programas. ALGOL 68 y Pascal no contienen características que satisfagan los puntos (a) y (b). La construcción class de SIMULA 67 satisface el punto (a) pero no el punto (b). Lenguajes más recientes como CLOU y Ada, proporcionan facilidades para definir tipos abstractos de datos que satisfagan ambas propiedades.

El concepto de tipo abstracto de dato se deriva de un principio más general: el ocultamiento de la información. Una parte de un programa que represente un tipo abstracto de dato es un ejemplo de un módulo que aplica la técnica de ocultamiento de la información (Sección 1.3). Los tipos abstractos de datos esconden los detalles de la representación y encaminan los accesos a los objetos abstractos a través de procedimientos. La representación está protegida contra cualquier intento de manipulación directa. Un posible cambio en la representación de un tipo abstracto de dato estará limitado a la parte del programa que describa dicha representación y no afectará al resto del programa.

2.3 ABSTRACCION DE CONTROL

Las estructuras de control describen el orden en el que se van a ejecutar las sentencias o los grupos de sentencias (unidades de programa). De la misma manera que las facilidades para la abstracción de datos, los mecanismos de abstracción de control pueden determinar la conveniencia o no de la utilización de un determinado lenguaje en una determinada área de aplicación. Las estructuras de control se pueden clasificar en estructuras de control a nivel de sentencia, es decir, aquellas que se usan para ordenar la ejecución de sentencias individualmente, y estructuras de control a nivel de unidad, es decir, aquellas que se usan para ordenar la ejecución de unidades de programa. A continuación se explica la evolución de las abstracciones de control con arreglo a esta clasificación.

2.3.1 Evolución de las Estructuras de Control a Nivel de Sentencia.

El hardware convencional proporciona únicamente dos mecanismos para gobernar el flujo de control sobre instrucciones individuales: la secuenciación y el salto. La secuenciación se realiza automáticamente incrementando el contador de programa después de la ejecución de cada instrucción. Este mecanismo permite que las instrucciones almacenadas en posiciones consecutivas de memoria sean ejecutadas una tras otra. El contador de programa también puede modificarse explícitamente por las instrucciones de salto, con el fin de realizar la transferencia de control a una posición específica distinta de la siguiente en secuencia.

En los lenguajes tipo ensamblador, las instrucciones a ejecutar consecutivamente se escriben una tras otra. Los saltos se representan por una instrucción jump. Por ejemplo, un bucle de N veces sobre un cierto conjunto de instrucciones precisa una inicialización, una modificación y una inspección de un contador (a menudo almacenado en un registro), como en el siguiente esquema (muchas máquinas incluyen una sola instrucción para modificar, inspeccionar y saltar).

```

set registro to N;
bucle: if el valor en el registro es cero jump to después;
<cuerpo del bucle>;
restaurar el valor del contador en el registro (si es
necesario) y decrementar en uno;
jump to bucle;
después: .....

```

Las estructuras de control a nivel máquina son difíciles de usar y propensas al error. Los programas resultantes son difíciles de leer y de mantener, porque esas estructuras no son naturales para los humanos. Los humanos organizan sus procesos de cálculo con arreglo a ciertos patrones ya establecidos, tales como la repetición o la selección entre diferentes elecciones. Por ejemplo, una manera mucho más natural de describir el programa anterior sería

```

hacer lo siguiente N veces
<cuerpo del bucle>

```

Con el fin de facilitar la programación y de promover un mejor estilo de la misma, a los lenguajes de alto nivel se les ha incorporado ciertas estructuras de control orientadas al usuario. Sin embargo, la mayoría de los lenguajes de alto nivel mantienen el salto incondicional en forma de sentencias goto, y por tanto, también soportan un estilo de programación de bajo nivel. Las sentencias goto son una fuente de oscuridad en la programación (ver la Sección 1.5.1). La controversia sobre la sentencia goto de principios de los 70 no proporcionó una solución definitiva al problema de qué estructuras de control deberían ser incluidas en un lenguaje de programación. En general, se podría decir que hay un amplio consenso acerca de que las sentencias goto sólo deberían ser utilizadas como una técnica para realizar estructuras de control "legítimas" si el lenguaje no incluye esas estructuras.

2.3.2 Evolución de las Estructuras de Control a Nivel de Unidad de Programa

2.3.2.1 Subprogramas y Bloques

Los lenguajes de programación proporcionan facilidades para agrupar sentencias que representen una acción abstracta en una adecuada unidad de programa. El ejemplo más usual y útil es el subprograma, que ha estado presente desde que aparecieron los primeros lenguajes de tipo ensamblador. Una definición de subprograma da un nombre a una cierta unidad de programa. Una llamada a un subprograma invoca a esa unidad de programa, es decir, transfiere el control a la unidad llamada, la cual devuelve el control al punto de llamada una vez haya acabado su tarea. Los convenios de paso de parámetros permiten a las unidades intercambiar información explícitamente.

Un ejemplo más simple de una unidad de programa es el bloque

de ALGOL 60, que soporta la agrupación de acciones, pero no la asignación de un nombre al grupo. Por tanto, un bloque no puede ser invocado explícitamente, sino que se ejecuta cuando se le encuentra durante la progresión normal de la ejecución.

Los subprogramas y los bloques, aunque éstos últimos en menor medida, son herramientas útiles para la estructuración de programas. En particular, los subprogramas soportan la distinción entre la definición de una acción abstracta (el cuerpo del subprograma) y su uso (la llamada al subprograma). Sin embargo, hay casos en los cuales el régimen de llamada a un subprograma y vuelta del mismo limita al programador. Varios lenguajes de reciente aparición reconocen esta limitación y proporcionan nuevas estructuras de control a nivel de unidad de programa.

2.3.2.2 Manejo de Excepciones

Los eventos o condiciones que una unidad de programa encuentra durante su ejecución pueden clasificarse en normales y excepcionales. Ejemplos de condiciones excepcionales pueden ser: un subprograma descubre que algunos valores de los parámetros pueden producir la ejecución de una división por cero, considerada ilegal; un asignador de memoria se sale fuera de la zona prevista para almacenamiento; se detecta un error de protocolo durante la recepción de un mensaje en una línea de transmisión, etc.

Para incrementar la legibilidad del programa y para aclarar las suposiciones del programador acerca de los eventos esperados y no esperados, es deseable poder dividir el programa en varias unidades. Algunas de ellas manejarán los eventos normales y podrán detectar la aparición de condiciones anómalas o excepcionales (llamadas excepciones). La aparición de una excepción transfiere automáticamente el control a una unidad apropiada, llamada manejador de excepciones, que tratará dicha excepción.

Los lenguajes de programación convencionales proporcionan poca ayuda en el tratamiento de las excepciones. Un subprograma que pueda provocar una excepción podría codificarse introduciendo un parámetro de vuelta adicional (p.ej., un entero) que indique un código de excepción (p.ej., 0 = no excepción, 1 = excepción número uno, etc.). La unidad que llame al subprograma podría comprobar el código de excepción después de cada llamada, y después, transferir el control al manejador de excepción apropiado si fuera necesario. Esta limitación en la potencia expresiva del lenguaje obliga a los programadores a exponer sus intenciones y suposiciones de una manera inadecuada, lo cual oscurece la lógica del programa.

PL/I fue el primer lenguaje de alto nivel en proporcionar el soporte idóneo para el manejo de excepciones. Posteriores diseños de lenguajes (Bliss, Mesa, CLU y Ada) han adoptado otras facilidades para el manejo de excepciones.

2.3.2.3 Corrutinas

Los subprogramas convencionales no pueden describir unidades de programa que avanzaan concurrentemente, como sucede en simulación discreta. Por ejemplo, la simulación de una partida de cartas con cuatro jugadores se podría hacer diseñando cuatro unidades de programa, una para cada jugador. Después de cada movimiento, cada unidad debería activar a la unidad que corresponda al siguiente jugador. Cuando una unidad es activada, debe reanudar su ejecución en el punto en que la dejó cuando transfirió el control a la siguiente unidad la última vez que lo hizo. Varios lenguajes disponen de una facilidad, la corrutina, para representar esta forma de ejecución intercalada. Cada corrutina que represente a un jugador de la partida de cartas debería tener la siguiente estructura general:

```
corrutina del jugador i
declaraciones de datos locales (p.ej. las cartas del jugador i);
while juego_no_acabado do
    seleccionar carta;
    jugar la carta;
    reanudar la ejecución de la corrutina correspondiente al
    siguiente jugador
end_of_do;
```

Los subprogramas convencionales están subordinados a sus llamantes tanto para iniciar su ejecución como para devolverles el control una vez finalice la misma. En la mayoría de los lenguajes, la operación de retorno origina la desaparición de los datos locales del subprograma llamado. En nuestro ejemplo, esto significaría que la información acerca de las cartas del jugador se perdería. A diferencia de los subprogramas, las corrutinas son unidades simétricas que se activan explícitamente unas a otras, y no devuelven control, sino que cada una reanuda a la siguiente. Cuando la corrutina A reanuda a la corrutina B, los datos locales de A se retienen. Una reanudación posterior de A podría continuar su ejecución desde el punto en que se abandonó anteriormente. En el ejemplo, la información acerca de las cartas de un jugador se retiene cuando otro jugador comienza a jugar.

2.3.2.4 Unidades Concurrentes

Las corrutinas son bastante adecuadas para simular actividades que se ejecutan de una forma intercalada. Sin embargo, en muchas aplicaciones es muy útil representar un sistema como un conjunto de unidades, llamadas unidades concurrentes, cuyo flujo avanza en paralelo (sean o no realmente ejecutadas en paralelo). Esta facilidad es particularmente importante en áreas tales como sistemas operativos. A la hora de describir unidades concurrentes, es necesario abstraerse de la arquitectura física de la máquina donde se van a ejecutar las unidades. La máquina podría ser un multiprocesador, con cada procesador dedicado a una sola unidad, o podría ser un único procesador multiprogramado. Esta posibilidad de ejecución sobre diferentes máquinas significa que la corrección de un sistema

concurrente no puede estar basada en ningún tipo de suposición acerca de la velocidad de ejecución de las unidades. De hecho, la velocidad puede diferir enormemente si cada unidad es ejecutada por un solo procesador dedicado a ella, o si un único procesador es compartido por varias unidades. Además, incluso si la arquitectura es conocida, es difícil diseñar un sistema de tal forma que su corrección dependa de la velocidad de ejecución de las unidades.

Las corrutinas son construcciones de lenguajes de bajo nivel para describir unidades concurrentes. Se pueden usar para simular paralelismo en un solo procesador, intercalando explícitamente la ejecución de un conjunto de unidades concurrentes. Por tanto describen, no un conjunto de unidades concurrentes, sino más bien una manera particular de compartir la UCP de un procesador con el fin de simular concurrencia. Muchos lenguajes más recientes poseen características especiales para tratar la concurrencia.

Aunque la concurrencia está siendo cada vez más un aspecto importante de los lenguajes, sus principales motivaciones y principios han provenido tradicionalmente del área de los sistemas operativos. El siguiente ejemplo puede ayudar aclarificar los problemas y conceptos básicos de la programación concurrente:

Supongamos que un cierto sistema contiene dos actividades concurrentes: un productor y un consumidor. El productor genera un conjunto de valores y los coloca en una zona de memoria de un cierto tamaño, N; el consumidor lee esos valores de la zona en el mismo orden en que fueron generados. Este modelo representa muchas funciones de los sistemas operativos, como puede ser la entrada y salida en un fichero de lectura/escritura. En la solución a este ejemplo que se muestra posteriormente, las unidades que representan las dos actividades se describen como unidades de programa cíclicas y sin fin.

+-----+	+-----+
unidad productora	unidad consumidora
+-----+	+-----+

```
repeat producir un elemento;
    añadir el elemento a la zona de memoria.
forever
```

+-----+	+-----+
unidad productora	unidad consumidora
+-----+	+-----+

```
repeat sacar un elemento de la zona de memoria;
    realizar algún cálculo sobre él
forever
```

Las dos unidades representan actividades que cooperan con el objetivo de alcanzar un fin común. Este fin común es transferir datos del productor (que podría estar leyéndolos de un dispositivo de entrada) al consumidor (que podría estar almacenándolos en un fichero). Sería deseable hacer que las unidades fueran insensibles a los cambios de velocidad de cualquiera de las dos actividades (p.ej., los cambios de velocidad del dispositivo de entrada). El mecanismo de

almacenamiento intermedio en una zona de memoria hace eso precisamente anulando dichas variaciones. Sin embargo, para garantizar una correcta cooperación, el programador debe asegurarse de que independientemente de la rapidez o de la lentitud con que avancen el productor y el consumidor, no habrá intentos de escribir en una zona llena o de leer de una zona vacía. Los lenguajes de programación concurrente proporcionan sentencias de sincronización que permiten al programador retener a una unidad cuando sea necesario con el propósito de coordinar la cooperación con otras unidades concurrentes. En el ejemplo, se debe retener al productor si intenta añadir un elemento cuando la zona ya está llena, hasta que el consumidor extraiga al menos un elemento. Igualmente, se deberá retener al consumidor si intenta extraer un elemento cuando la zona esté vacía, hasta que el productor añada al menos un nuevo elemento.

Otra necesidad más sutil de sincronización puede aparecer cuando dos actividades puedan acceder legalmente a la zona de memoria. Por ejemplo, supongamos que las operaciones añadir y sacar actualizan el valor de t (número total de unidades almacenadas) realizando (1): $t := t+1$ y (2): $t := t-1$, respectivamente. Supongamos también que (1) y (2) se representan mediante:

leer t y escribirlo en un registro privado;
actualizar el valor almacenado en dicho registro;
escribir en t el valor almacenado en el registro privado

donde actualizar es el "incrementar en uno", o el "decrementar en uno", de (1) y (2) respectivamente. Las acciones (1) y (2) son instrucciones máquina indivisibles, en el sentido de que una vez que cualquiera de esas acciones comience a ejecutarse, está garantizado que acabarán antes de que otra operación empiece. Esto contrasta con sus respectivas representaciones ya que en ellas la ejecución de sus acciones constituyentes puede estar intercalada. Una de las múltiples secuencias posibles de acciones intercaladas podría ser como sigue: la primera acción de (1); la primera acción de (2); la segunda acción de (1); la segunda acción de (2); la tercera acción de (1) y finalmente la tercera acción de (2).

Dado que (1) y (2) no son acciones indivisibles, es fácil verificar que si m es el valor de t antes de la ejecución concurrente de un par (añadir, sacar), el valor de t después de la ejecución puede ser m , $m+1$, o $m-1$, cuando en realidad el único valor correcto sería el primero. Para garantizar la corrección, el programador se debe asegurar de que (2) no empieza mientras (1) se está ejecutando, y viceversa, es decir, que (1) y (2) se deben ejecutar en exclusión mutua, como si fueran operaciones "indivisibles".

Estamos ahora en situación de determinar algunos requisitos que deben cumplir las abstracciones que necesitamos para tratar la concurrencia, así como las construcciones del lenguaje que se pueden utilizar para definir tales abstracciones.

Un sistema concurrente debería verse como un conjunto de procesos, cada uno de los cuales está representado por una unidad de programa. Los procesos son concurrentes si sus ejecuciones se pueden solapar (conceptualmente) en el tiempo, es decir, si el comienzo de un proceso puede darse cuando el proceso que se ejecutaba previamente todavía no ha concluido. Los procesos P_1 , P_2 , ..., P_n son disjuntos si describen actividades que nunca interactúan una con otra, es decir, no acceden a ningún objeto compartido. El resultado de la ejecución de un proceso es independiente de otros procesos paralelos. Sin embargo, muy a menudo los procesos son interactivos. La interacción es debida a una de las dos razones siguientes:

Competición. Los procesos compiten en el acceso a un cierto recurso compartido, que debe ser utilizado en exclusión mutua (p. ej., una impresora).

Cooperación. Los procesos cooperan para alcanzar un objetivo común.

Los procesos interactúan correctamente sólo si existe una cierta relación de precedencia entre sus acciones elementales, es decir, ciertas acciones deben preceder a otras ciertas acciones. Tal relación define una ordenación parcial entre acciones. Por ejemplo, en el caso del par productor/consumidor, si $P_j(C_j)$ representa la producción (consumición) del elemento número j , una correcta cooperación requiere ("-->" debe leerse "precede a")

$C_j \rightarrow P_{j+n} \text{ y } P_j \rightarrow C_j$ para todo j

Una correcta competición requiere que no se puedan ejecutar dos actualizaciones de la variable compartida t al mismo tiempo. Si $IT_j(DT_j)$ representa el incremento (decremento) número j de la variable t realizado por el productor (consumidor), la competición correcta requiere que

$IT_j \rightarrow DT_k \text{ o } DT_j \rightarrow IT_k$ para todo j y k

Los lenguajes de programación concurrente proporcionan facilidades para la definición de procesos y sentencias adecuadas de sincronización que hacen cumplir la necesaria ordenación parcial de acciones. Las corutinas, tal como se ha explicado previamente, pueden utilizarse para simular la concurrencia en un solo procesador, pero no proporcionan el soporte de abstracción más adecuado para este propósito. Especifican en demasía el sistema, ya que muestran explícitamente cuándo un proceso recoge el control de otro. En otras palabras, imponen una ordenación total de acciones, cuando una ordenación parcial es suficiente y perfectamente adecuada para la descripción del sistema.

2.4 CORRECCIÓN

La corrección es uno de los requisitos básicos en la programación. Hay dos enfoques diferentes en la producción de programas correctos. El primero, corrección de errores, concierne a programas ya escritos, y consiste en el tratamiento adecuado de los mismos cada vez que aparezca un síntoma de error. El segundo, prevención de errores, consiste en tratar de desarrollar programas que son correctos desde el principio.

Obviamente, no se desarrolla ningún programa con total despreocupación de su corrección, y tampoco se desarrolla ningún programa tan cuidadosamente que esté libre de posibles errores. Sin embargo, es verdad que la mayoría del coste de la producción de los programas es debida a la corrección de errores y que la adopción de herramientas adecuadas y sistemáticas en la fase del diseño puede ayudar a prevenir la introducción de errores.

Un camino para favorecer la producción de programas correctos es hacer que el esfuerzo de diseño y de codificación sea fácilmente manejable, de tal forma que estemos seguros del comportamiento deseado del sistema. Las abstracciones en datos y en control, tal como se ha visto en las Secciones 2.2 y 2.3, son poderosos mecanismos para dominar la complejidad del diseño de un programa. Las construcciones de un lenguaje que permitan la correspondencia entre las abstracciones de diseño y las estructuras del programa promueven la producción de programas correctos y bien estructurados. Dichas estructuras del programa hacen posible acabar con la compleja tarea de seguir la lógica de un objeto grande, el programa completo, para hacerlo sobre objetos abstractos más pequeños, más manejables y autónomos en gran medida. Además, las unidades de programa deben ser fáciles de leer y entender, de tal forma que los posibles errores puedan ser localizados fácilmente. En los lenguajes se han identificado una cierta cantidad de características perjudiciales, es decir, características que hacen que sea difícil seguir la lógica de los programas, empezando con el conocido caso de la sentencia *goto*. (En el Capítulo 6 se presenta un análisis de estas características y unas cuantas soluciones alternativas propuestas por los diseñadores de lenguajes).

Sin embargo, incluso programas diseñados sistemáticamente pueden contener errores y es por ello necesario idear estrategias para aislar y eliminar dichos errores. Un esquema sólido para una certificación sistemática de los programas, a diferentes niveles, lo proporciona el propio lenguaje de programación. A un nivel más bajo están las comprobaciones de consistencia, que verifican que los programas se adaptan a la definición del lenguaje. Por ejemplo, los programas deben estar sintácticamente bien formados (es decir, deben ser consistentes con las reglas sintácticas del lenguaje); las variables del programa se deben utilizar de una manera consistente con su tipo; a los subprogramas se les debe llamar con parámetros reales que sean consistentes, en número y tipo, con los parámetros formales que aparecen en la cabecera del subprograma.

Las comprobaciones de consistencia realizadas antes de la ejecución del programa se llaman comprobaciones estáticas; las que se realizan durante la ejecución se llaman comprobaciones en tiempo de ejecución (o dinámicas). Las comprobaciones de sintaxis realizadas por un traductor son un ejemplo de comprobaciones estáticas. Las comprobaciones de tipo son otro ejemplo de comprobaciones de consistencia que precisan la mayoría de los lenguajes, y que a menudo se realizan estáticamente. El siguiente fragmento de un programa en Pascal ilustra estos puntos.

```
var x,y: integer;
z: char;

x := (((x+y)*5+x*y);
if x<0 then y := z;
```

La sentencia de asignación contiene un error de sintaxis, ya que le falta un paréntesis cerrado. La sentencia *if* contiene un error de tipo porque a una variable de tipo entero(y) se le asigna una de tipo carácter (z). Los dos errores son detectables estáticamente.

Aunque cualquier error detectable estáticamente también podría ser detectado en tiempo de ejecución, sería poco aconsejable retrasar la comprobación de dicho error al tiempo de ejecución por dos razones. Una es que las posibles fuentes de error sólo pueden ser detectadas en tiempo de ejecución si se dispone información de todos los datos que podrían producir dicho error, por ejemplo, el fragmento de programa incorrecto escrito anteriormente señalaría el error de tipo sólo si el valor de x fuera negativo. Y en segundo lugar, la comprobación dinámica ralentiza la ejecución del programa.

Sin embargo, no todos los lenguajes permiten realizar totalmente la comprobación de tipos antes de la ejecución. Por ejemplo, en APL el tipo de una variable se determina por el valor que dicha variable tenga, y por tanto, puede cambiar durante la ejecución del programa, como muestra el siguiente fragmento de programa.

```
A <-- 'CADENA'
.
.
.
A <-- 3.77
```

En consecuencia, la suma de un valor numérico a A sólo es correcta si el valor de A es un número, y no una cadena de caracteres. Y en general, esto sólo puede ser comprobado en tiempo de ejecución.

Podemos resumir diciendo que la corrección puede ser mejorada por un lenguaje cuya definición precise comprobaciones exhaustivas en los programas, y mucho mejor si dichas comprobaciones se pueden hacer estáticamente. Los primeros lenguajes no reconocían la necesidad de características que soportarán la corrección de los programas. Sin embargo, los lenguajes más recientes han sido diseñados con el objetivo de soportar comprobaciones exhaustivas.

Incluso contando con la presencia de comprobaciones estáticas, el programador puede producir programas que no se ajusten a las especificaciones. En este caso, el programa se modifica gradualmente hasta que pueda ser certificado. La herramienta tradicional para certificar programas consiste en poner a prueba el programa sometiéndole a un conjunto de datos de entrada, tomado de los documentos de especificación, enfocado a poner de manifiesto algunos posibles errores del programa, y observar si la salida producida por el mismo se ajusta al comportamiento esperado. Los resultados incorrectos son síntomas de errores que deben ser extraídos del programa. Sin embargo, los resultados correctos no implican la corrección del mismo; todo lo que se puede afirmar es que el programa se comporta correctamente para esos datos particulares de entrada. Así, siguiendo un comentario de Dijkstra, podemos decir que las pruebas se pueden usar para demostrar la presencia de errores pero no su ausencia.

Esta deficiencia intrínseca de la prueba de los programas ha impulsado una gran cantidad de investigación en el campo de la verificación de los programas. Al contrario que la prueba, la verificación intenta comprobar la corrección de un programa independientemente de su ejecución. Verificar un programa, en sentido estricto, significa probar que la implementación del mismo es consistente con su especificación. La especificación a su vez, es entendida aquí como la descripción formal y precisa de lo que el programa debe hacer.

La verificación de un programa se puede hacer manualmente, pero en este caso la confianza en la corrección del programa depende a su vez de la confianza en la corrección del propio proceso manual. Se han desarrollado sistemas automáticos para verificación de programas, pero no hay experiencia real y práctica de su uso en la verificación de grandes sistemas. Un ejemplo se comenta en (Walker y otros, 1979). Los autores de este experimento llegaron a la conclusión de que "no parece haber razones técnicas, aparte de la necesidad de un sistema adecuado (asistido por ordenador) de verificación ... para que los métodos de prueba de programas no puedan ser empleados en el desarrollo de software donde una operativa correcta es crítica ... Sin embargo, las técnicas actuales todavía no son adecuadas para utilizarse de una forma generalizada".

Ni la prueba ni la verificación son apropiadas para proporcionar la respuesta final y definitiva a la necesidad de tener programas correctos. La verificación es teóricamente un sistema más válido que la prueba. Sin embargo, en las primeras fases del desarrollo de un programa, cuando lo normal es que todavía haya muchos errores, la prueba puede ser más económica que la verificación. En otras palabras, las dos técnicas se complementan la una a la otra, y juntas proporcionan una mayor seguridad de corrección que la que puedan ofrecer cada una por si sola. Desgraciadamente todavía está por descubrir un método utilizable en la práctica que esté teóricamente bien fundamentado y que agrupe una mezcla de los dos enfoques.

2.5 PROGRAMACION A GRAN ESCALA

La producción de grandes sistemas requiere a menudo la coordinación de actividades de una gran cantidad de personas. Los programas se forman a partir de colecciones de módulos individuales que se agrupan, y dichos módulos pueden estar escritos por personas diferentes y codificados en distintos lenguajes. Los programas muy grandes ("programas casi imposibles", según [Yourdon, 1975]) pueden tener del orden de un millón de sentencias fuente, pueden estar escritos por centenas de programadores durante un periodo de varios años, y pueden constar de varios centenares de módulos con complejas interacciones entre ellos y con otros sistemas desarrollados separadamente. Dentro de esta categoría de programas están las aplicaciones gubernamentales y militares, los grandes proyectos para bancos, los sistemas de manejo de información, y muchas otras aplicaciones. Una gestión apropiada del diseño, programación, certificación y mantenimiento de tales sistemas es una tarea enorme, que implica problemas que van desde la sociología de las relaciones humanas a los métodos para la producción de programas.

A un nivel más bajo de tamaño y de complejidad, los proyectos que implican la producción de programas grandes (con varios miles de sentencias fuente, escritas por entre 5 y 20 programadores durante un periodo de dos o tres años) están siendo cada vez más comunes. La gestión del diseño y de la producción de programas de este tipo requiere métodos y herramientas adecuadas con el fin de controlar su complejidad.

El ciclo de vida total de la producción de programas en el caso de sistemas grandes puede ser bastante diferente del correspondiente a los programas de tamaño pequeño. En (DeRemer y Kron, 1976) se argumenta que "la programación a gran escala es una actividad intelectual esencialmente distinta y diferente de la de construir módulos individuales", es decir, de lo que podríamos denominar programación a pequeña escala. En consecuencia, llegan a la conclusión de que "se deberían utilizar lenguajes esencialmente distintos para las dos actividades".

Hemos supuesto implícitamente que los sistemas grandes están

construidos a partir de componentes individuales, llamados módulos. La modularidad, de hecho, es generalmente reconocida como el único medio disponible para dominar la complejidad del diseño y la realización de sistemas grandes y complejos. Sin embargo, "modularidad" es una palabra ambigua que a menudo es utilizada para describir varias propiedades de los programas. En muchos casos, la modularidad se define en términos del tamaño de las unidades de programa que comprenden un sistema. Así, por ejemplo, hay organizaciones que adoptan convenios de producción tales como "Cada subrutina FORTRAN debe estar contenida en una página de listado". Es evidente sin embargo, que las restricciones que se aplican únicamente al tamaño de los módulos no mejoran la calidad de los programas en un sentido real, ya que dividir un texto largo de un programa en una secuencia de piezas más pequeñas no hace que el programa sea mejor.

Una noción más útil de modularidad está basada en el concepto de independencia. La idea es que cada módulo debería ser entendido y, en lo posible, realizado independientemente del resto de los módulos del sistema. Cada módulo debería realizar una única y simple función conceptual del sistema, y las restricciones acerca del tamaño del módulo son una consecuencia del método seguido en el proceso de diseño. El ocultamiento de la información como principio de diseño favorece la producción de módulos altamente independientes. De hecho, las decisiones de diseño internas a un módulo se esconden y no afectan a la corrección de la cooperación entre los módulos. Una vez que la interconexión de los mismos ha sido (cuidadosamente) diseñada, su desarrollo puede ser independiente, se pueden almacenar en una librería y posteriormente se pueden ensamblar para construir un programa.

El objetivo de diseñar los programas modularmente ejerce una fuerte influencia sobre los lenguajes de programación. Estos lenguajes deberían proporcionar facilidades para la definición de módulos y para el ocultamiento de la información. E igualmente deberían hacerlo para poder estructurar fácilmente una colección de módulos en un sistema único. Finalmente, debería ser posible desarrollar y certificar módulos separadamente. Algunos diseños recientes de lenguajes (p. ej., Ada) han estado enormemente influidos por estos conceptos.

La necesidad de producir a gran escala software complejo y fiable también ha influido en el desarrollo de un cierto número de herramientas que pueden ayudar al programador a diseñar, codificar, certificar y mantener programas individuales. Como ya apuntamos en el Capítulo 1, los lenguajes de programación y los métodos para el desarrollo de programas, considerados aisladamente, no son suficientes para incrementar significativamente la productividad del programador. En cambio, la combinación de un lenguaje adecuado con un número de herramientas sensibles al lenguaje, fáciles de usar, integradas y potentes, es decir, un paquete de facilidades para el desarrollo, puede llegar a ser el factor clave en la mejora de la calidad de los programas. El primer paso en esta dirección se ha dado con

sistemas como el UNIX, que fue desarrollado por los Laboratorios Bell para la familia de ordenadores de DEC. Sin embargo, el desarrollo de un sistema consistente y completo para la fabricación de programas es todavía un problema abierto a la investigación.

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

(Sammet 1969) es un resumen de un gran número de lenguajes de programación y proporciona una amplia visión de la evolución en este campo. (Wegner 1978) estudia la evolución de los lenguajes hasta 1975. Los debates de la conferencia sobre la Historia de los Lenguajes de Programación de ACM SIGPLAN (ACM-SIGPLAN 1978) supusieron un primer intento de estudiar desde una perspectiva histórica los esfuerzos de diseño de lenguajes realizados hasta entonces. Varias ponencias de (ACM-SIGPLAN 1978) resaltaban explícitamente la fuerte influencia ejercida por el hardware sobre las abstracciones proporcionadas por los lenguajes (véase, por ejemplo, la ponencia de Backus sobre FORTRAN).

La evolución del concepto de abstracción en los lenguajes se trata en (Guarino 1978).

En (Wegner 1979) el autor presenta una amplia perspectiva de las ramas de investigación en el área de los lenguajes.

Una solución al Ejercicio 2.6 se puede encontrar en (Conway 1963), lo que supuso el origen de las corrutinas.

Este capítulo ha subrayado los conceptos de los lenguajes y su evolución. Del Capítulo 3 al 8 se presentará un análisis detallado y una evaluación crítica de las soluciones adoptadas por varios lenguajes. En el apartado de Ampliación Bibliográfica de los Capítulos 4 al 7 se pueden encontrar más sugerencias de lecturas posteriores y notas bibliográficas sobre los puntos esbozados en las Secciones 2.2 a la 2.5

Ejercicios

2.1 Las abstracciones de un lenguaje de programación se pueden clasificar con arreglo a su facilidad para soportar la escritura de programas. A menudo, cuanto más sencillas son las abstracciones, más difíciles representarlas en un ordenador.

Con arreglo a esta caracterización, comparar la sentencia IF de FORTRAN con la IF-THEN-ELSE de los lenguajes tipo ALGOL.

2.2 Dado que en un ordenador sólo se puede representar un número finito de valores, el tipo de dato entero soportado por el hardware no corresponde exactamente con la noción matemática

- de entero. ?Qué sucede en su ordenador cuando el resultado de una expresión entera está fuera del rango aceptable de valores?
- 2.3 Como parte de un gran proyecto de programación, a usted se le asigna el diseño y la realización de una estructura en forma de cola, llamada ALMACEN, para guardar y recuperar valores enteros. Los valores se extraen de la cola según una disciplina "primero en entrar/primer en salir". La cola debe ser capaz de detectar intentos de inserción en cola llena y de extracción de cola vacía. Ya que las especificaciones del proyecto todavía no son firmes, usted debe diseñar su programa de tal forma que sea fácilmente modificable. Por ejemplo, la capacidad máxima o la organización del almacenamiento de la cola podrían cambiarse más adelante.
- Producir la especificación de ALMACEN tan abstractamente como sea posible, y después diseñar una realización concreta.
 - Codificar una realización concreta en cualquier lenguaje de programación.
 - Discutir la modificabilidad de la solución adoptada y la forma en que el lenguaje dificultó o ayudó a alcanzarla.
- 2.4 El fragmento de programa tipo ALGOL
- ```
while a>b do
 ...
end-of-do
```
- se puede reescribir usando saltos (condicionales e incondicionales). Comparar brevemente los dos enfoques en términos de legibilidad y de facilidad de escritura.
- 2.5 Un programa debe leer un valor entero e imprimir "sí" si el valor es cero y "no" en cualquier otro caso. Escribir una solución usando la sentencia IF de FORTRAN pero sin GOTO's. ?Es muy natural su solución? ?Puede usted escribir una solución mejor usando la sentencia IF de los lenguajes tipo ALGOL?
- 2.6 Este ejercicio ilustra el uso de corrutinas. Diseñar un programa que lea caracteres de tarjetas perforadas y los imprima. Cada aparición de un par de asteriscos ("\*\*") en la entrada debe ser reemplazada por el único carácter "†" en la salida. El resto de caracteres deben ser simplemente copiados. Su solución debe consistir en dos corrutinas: una de entrada que proporcione el siguiente carácter, y otra de salida que imprima caracteres.
- 2.7 Los valores de dos variables, vi y v2, se intercambian en cada uno de dos procesos concurrentes, P1 y P2. El intercambio se realiza mediante la siguiente secuencia de operaciones indivisibles a nivel máquina.
- ```
Cargar vi en el registro R1.
Cargar v2 en el registro R2.
Almacenar el valor del registro R1 en v2.
Almacenar el valor del registro R2 en vi.
```
- ?Cuál es el efecto de la ejecución concurrente si P1 y P2 no ejecutan el intercambio en exclusión mutua?
- 2.8 Dar ejemplos de comprobaciones estáticas y dinámicas soportadas por el lenguaje de programación que más le guste.
- 2.9 La prueba exhaustiva es la prueba de un programa para todos los valores posibles de sus variables de entrada. Dar argumentos (y ejemplos) que demuestren que la prueba exhaustiva es impracticable.
- 2.10 ?Cuáles son los criterios que usted sigue para seleccionar datos para la prueba de sus programas? ?Qué nivel de seguridad tiene usted acerca de la corrección de su programa después de la prueba?
- 2.11 ?Qué lenguaje de los que usted conoce es más conveniente para la programación a gran escala y por qué?
- 2.12 ?Ha sido usted alguna vez miembro de algún equipo de trabajo sobre un programa de tamaño sustancial? Si es así, ?con qué tipo de dificultades tuvo que enfrentarse que no se hubiera encontrado a la hora de escribir un programa más simple usted solo?

**INTRODUCCION
A LA SEMANTICA
DE LOS LENGUAJES
DE PROGRAMACION**

Un lenguaje de programación es una notación formal para describir algoritmos que serán ejecutados por un ordenador. Al igual que todas las notaciones formales, un lenguaje tiene dos componentes: sintaxis y semántica.

La sintaxis es un conjunto de reglas formales que especifican la composición de los programas a base de letras, dígitos y otros caracteres. Por ejemplo, las reglas de sintaxis pueden especificar que cada paréntesis abierto debe llevar su correspondiente paréntesis cerrado en las expresiones aritméticas, y que dos sentencias deben ir siempre separadas por punto y coma. Las reglas semánticas especifican el significado de cualquier programa sintácticamente válido, escrito en ese lenguaje. Dicho significado puede expresarse mediante la correspondencia entre cada construcción del lenguaje en un dominio cuya semántica es conocida. Por ejemplo, una manera de describir la semántica de un lenguaje es dando una descripción en español de cada construcción del lenguaje. Desde luego, esta descripción mantiene la informalidad, ambigüedad y verbosidad del lenguaje natural, pero puede darnos una visión bastante intuitiva del lenguaje.

En el presente capítulo abordaremos la semántica desde el punto de vista funcional y la describiremos especificando el comportamiento de un procesador abstracto que ejecuta programas escritos en un cierto lenguaje. Esta caracterización semántica de un lenguaje podría hacerse utilizando una notación formal y rigurosa. Sin embargo, lo haremos de una manera más tradicional e informal, pues de este modo, los programadores lo comprenden más intuitiva y fácilmente, a la vez que nos da una visión a más alto nivel de los problemas que se presentan a la hora de implementar un lenguaje.

En este capítulo se describen los conceptos básicos de los lenguajes de programación y se hace una introducción a las características semánticas básicas necesarias para su comprensión. Utilizaremos FORTRAN, la familia de los lenguajes de tipo ALGOL, y APL a modo de ejemplos ilustrativos de las características fundamentales (sin asumir por ello el conocimiento previo de estos lenguajes). Los principios y técnicas que se describen son generales y aplicables a los lenguajes modernos, como ya veremos más adelante.

No entraremos a discutir en detalle la sintaxis de los lenguajes. Nos dedicaremos a las posibilidades de un lenguaje, que no estén afectadas por la sintaxis. Sin embargo, debemos insistir en que la sintaxis de un lenguaje tiene una gran influencia en la facilidad con que los programadores emplearán este lenguaje y en la legibilidad de los programas escritos con él.

El presente capítulo está organizado de la siguiente manera: La Sección 3.1 presenta dos estrategias diferentes para procesar un lenguaje de programación: interpretación y traducción. La Sección 3.2 describe el concepto general de ligadura, que se utilizará para describir algunas propiedades semánticas de los

lenguajes. La Sección 3.3 se dedica a las variables de los programas y sus atributos básicos. La Sección 3.4 se refiere a las unidades de programa y su representación en tiempo de ejecución. Las Secciones 3.5 a 3.7 detallan todos estos conceptos en el caso de FORTRAN, lenguajes de tipo ALGOL y APL.

3.1. PROCESO DE UN LENGUAJE

A pesar de ser teóricamente posible la construcción de ordenadores de propósito especial que ejecuten directamente programas escritos en un lenguaje concreto, los ordenadores actuales solamente ejecutan un lenguaje de muy bajo nivel, el lenguaje máquina. Estos lenguajes máquina están diseñados basándose en la velocidad de ejecución, coste de realización y flexibilidad para construir sobre ellos nuevos niveles de software. Por otra parte, los lenguajes se diseñan generalmente sobre la base de su fiabilidad y facilidad de programación. Surge entonces el problema de cómo ejecutar un lenguaje de alto nivel en un ordenador cuyo lenguaje máquina sea muy distinto y de un nivel mucho más bajo.

Veamos las dos alternativas que existen para la implementación: interpretación y traducción.

3.1.1 Interpretación

Con esta solución se ejecutan directamente las acciones que implican las sentencias del lenguaje. Generalmente existe un subprograma para cada acción posible (escrito en lenguaje máquina) que ejecuta la acción. Por lo tanto, para interpretar un programa habrá que llamar a los subprogramas en la secuencia apropiada.

Para ser más exactos, un intérprete es un programa que ejecuta repetidamente la secuencia siguiente:

1. Obtener la sentencia siguiente.
2. Determinar las acciones que han de ser ejecutadas.
3. Ejecutar las acciones.

Esta secuencia es muy similar al conjunto de acciones llevadas a cabo por un ordenador tradicional:

1. Extraer la siguiente instrucción, es decir, la instrucción cuya dirección aparece en el contador del programa.
2. Avanzar el contador del programa, es decir, establecer la dirección de la siguiente instrucción a extraer.
3. Decodificar la instrucción.

4. Ejecutar la instrucción.

Esta similitud nos muestra que podemos considerar la interpretación como una simulación en un ordenador anfitrión de una máquina de propósito especial, cuyo lenguaje máquina es el lenguaje de alto nivel.

3.1.2 Traducción

Con esta solución, un programa escrito en lenguaje de alto nivel se traduce a una versión equivalente en lenguaje máquina antes de ser ejecutado. Esta traducción se hace en varias etapas. Los subprogramas pueden traducirse primero a ensamblador y éste a su vez se traduce a código máquina reubicable; a continuación se fusionan las unidades de código reubicable en una única unidad reubicable y finalmente se carga en memoria principal el programa entero como código máquina ejecutable. Los traductores que se utilizan en cada una de estas etapas tienen nombres específicos: compilador, ensamblador, editor de enlaces y cargador respectivamente.

En algunos casos, la máquina en la que se lleva a cabo la traducción (máquina anfitriona) no es la misma que la que ha de ejecutar el código traducido (máquina destino). Este tipo de traducción recibe el nombre de traducción cruzada. Los traductores cruzados son la única solución viable en el caso de que la máquina destino sea demasiado pequeña para soportar al traductor.

La interpretación pura y la traducción pura son dos extremos. En la práctica ocurre que muchos lenguajes se implementan a base de la combinación de ambas técnicas. Un programa se puede traducir a un código intermedio que será a su vez interpretado. Este código intermedio puede ser, por ejemplo, una simple representación formateada del programa original, del que se ha eliminado la información no significativa (p.ej., comentarios y espacios en blanco) y en la que los componentes de cada sentencia se han almacenado con formato fijo, con objeto de simplificar la subsiguiente decodificación de las instrucciones. En este caso, podemos decir que la solución es básicamente interpretativa. Alternativamente, el código intermedio puede ser el código máquina (bajo nivel) para una máquina virtual que a su vez será interpretada por software. Esta solución se apoya principalmente en la traducción y se puede aplicar en la generación de un código transportable o, lo que es lo mismo, un código que sea más fácil de transferir a máquinas diferentes que el código en lenguaje máquina.

En el caso de una solución puramente interpretativa, la ejecución de una sentencia puede llevar consigo un proceso de decodificación bastante complicado para determinar las operaciones que han de ser ejecutadas, así como sus operandos. En la mayoría de los casos, cada vez que aparece una sentencia se hace un proceso idéntico. Por lo tanto, si la sentencia aparece

en una parte del programa que ha de ejecutarse con cierta frecuencia (p.ej., un bucle interno) resulta que estos procesos idénticos de decodificación afectarán considerablemente a la velocidad de ejecución. Por el contrario, la traducción pura genera código máquina para cada sentencia de alto nivel, con lo cual, el traductor decodifica solamente una vez cada sentencia de alto nivel. Ocurre con frecuencia que las partes ya utilizadas son decodificadas muchas veces en su representación en lenguaje máquina y dado que esto lo hace el hardware muy eficazmente, resulta que la traducción pura suele ahorrar tiempo de proceso en comparación con la interpretación pura. Por otro lado, la interpretación pura suele ahorrar memoria. En la traducción pura, cada sentencia en lenguaje de alto nivel puede originar decenas o centenas de instrucciones máquina. En una solución puramente interpretativa, las instrucciones de alto nivel se dejan en su forma original y las instrucciones necesarias para su ejecución quedan almacenadas en un subprograma del intérprete. El ahorro de memoria es evidente en el caso de un programa extenso y que utilice la mayoría de las sentencias del lenguaje. Por el contrario, si todos los subprogramas del intérprete se almacenaran en la memoria principal durante la ejecución, resultaría que el intérprete desperdiciaría espacio para programas pequeños que sólo utilizaran unas cuantas sentencias del lenguaje.

3.2 CONCEPTO DE LIGADURA

Los programas manejan entidades tales como variables, subprogramas, sentencias, etc. Las entidades de un programa tienen ciertas propiedades llamadas atributos. Por ejemplo, una variable tiene un nombre, un tipo y un área de almacenamiento donde se guarda su valor; un subprograma tiene a su vez un nombre, parámetros formales de un cierto tipo y ciertas convenciones para pasar dichos parámetros; por último, una sentencia tiene acciones asociadas. Antes de procesar una entidad hay que especificar sus atributos. Recibe el nombre de ligadura la especificación de la naturaleza exacta de un atributo. La información de la ligadura de cada entidad se halla en una zona llamada descriptor.

El concepto de ligadura es de la mayor importancia a la hora de definir la semántica de los lenguajes de programación. Los lenguajes difieren entre sí por el número de entidades que son capaces de manejar, por el número de atributos que han de ser ligados a las entidades y por el momento en el que se producen dichas ligaduras (tiempo de ligadura). Se dice que una ligadura es estática si ha quedado establecida antes de la ejecución y ya no puede cambiarse. Por el contrario, será dinámica si se establece durante el tiempo de ejecución y puede cambiarse de acuerdo con algunas reglas especificadas en el lenguaje.

Los conceptos de ligadura y tiempo de ligadura ayudan en gran medida a clarificar muchos aspectos semánticos de los lenguajes de programación. A continuación nos serviremos de estos

conceptos para ilustrar el concepto de variable.

3.3 VARIABLES

Los ordenadores convencionales se basan en la idea de una memoria principal que consiste en celdas elementales identificadas cada una mediante una dirección. Cada celda contiene un valor que puede ser leído y/o modificado. La modificación consiste en reemplazar un valor por otro nuevo. El hardware nos permite acceder a las celdas de una en una. Salvo raras excepciones, los lenguajes de programación pueden ser considerados como abstracciones a diferentes niveles del comportamiento de los ordenadores convencionales. En concreto, introducen el concepto de variable como una abstracción del concepto de celda de memoria, y el concepto de sentencia de asignación como abstracción de la modificación destructiva del contenido de una celda.

En este capítulo y en los siguientes nos limitaremos a considerar estos lenguajes de programación convencionales "basados en la asignación". En el Capítulo 8 veremos lenguajes alternativos que soportan un estilo funcional de programación.

Una variable se caracteriza por un nombre y por cuatro atributos básicos: ámbito, tiempo de vida, valor y tipo. El nombre se utiliza para identificar y hacer referencia a la variable. Algunos lenguajes permiten la existencia de variables sin nombre; más adelante daremos algunos ejemplos de estas variables llamadas anónimas. A continuación se detallan cada uno de estos cuatro atributos y las diferentes normas adoptadas por los lenguajes para ligar atributos a variables.

3.3.1 Ámbito de una Variable

El ámbito de una variable es el conjunto de sentencias del programa en las que dicha variable es conocida y por lo tanto manipulable. Se dice que una variable es visible dentro de su ámbito e invisible fuera de él. Las variables pueden ir ligadas a un ámbito de manera estática o dinámica.

La ligadura de ámbito estático define el ámbito de una variable en términos de la estructura léxica de un programa, esto es, cada referencia a una variable está estéticamente ligada a una declaración de variable concreta (implícita o explícita). Las reglas del ámbito estático han sido adoptadas por la mayoría de los lenguajes que veremos en este libro.

La ligadura de ámbito dinámico define el ámbito de una variable en términos de ejecución del programa. Normalmente, cada declaración de variable ejerce su influencia sobre todas las sentencias que se ejecutan a partir de dicha declaración, hasta que se encuentra una nueva declaración para una variable con el mismo nombre. APL, LISP y SNOBOL-4 son ejemplos de lenguajes con

reglas de ámbito dinámico.

Las reglas de ámbito dinámico son bastante fáciles de implementar, pero presentan algunas desventajas en cuanto a la disciplina de programación y eficacia de la implementación. Los programas resultan difíciles de leer dado que la identidad de la declaración a la que está ligada una variable dada, depende del punto concreto de la ejecución, de manera que no se puede determinar estáticamente.

3.3.2 Tiempo de Vida de una Variable

El tiempo de vida de una variable es el intervalo de tiempo en el que un área de memoria está ligada a la variable. Este área se utiliza para contener el valor de la variable. Utilizaremos el término objeto de datos (o simplemente objeto) para denominar el área de memoria y el valor conjuntamente.

La acción de conseguir un área de memoria para una variable se denomina asignación de memoria. En algunos lenguajes esta asignación se lleva a cabo antes del tiempo de la ejecución del programa (asignación estática). En otros lenguajes se hace durante la ejecución (asignación dinámica), bien mediante una petición explícita del programador por medio de una sentencia de creación, o bien automáticamente al entrar en el ámbito de la variable. En las Secciones 3.4 a 3.7 podemos encontrar información detallada sobre este tema.

3.3.3 Valor de una Variable

El valor de una variable se representa en forma codificada en el área de memoria ligada a dicha variable. Esta representación codificada se interpreta de acuerdo con el tipo de la variable.

En algunos lenguajes, el valor de una variable puede ser una referencia (puntero) a un objeto. En estos lenguajes se puede acceder a un objeto por medio de una cadena de referencias (o camino de acceso) de longitud arbitraria. Dos variables comparten un objeto si cada una de ellas tiene un camino de acceso a dicho objeto. Un objeto compartido que se haya modificado por medio de un camino de acceso, hace que esta modificación sea conocida por todos los posibles caminos de acceso restantes. El compartir objetos se utiliza para ahorrar espacio en memoria, pero puede dar lugar a programas difíciles de leer, dado que el valor de las variables puede ser modificado aun cuando no hayan sido referenciadas. Una referencia es el medio básico para acceder a variables anónimas.

La ligadura entre una variable y el valor contenido en su área de memoria es generalmente dinámica, dado que se puede modificar el valor mediante una operación de asignación. Una asignación del tipo $b := a$ hace que se lleve una copia del valor

de a al área de memoria ligada a b .

Algunos lenguajes permiten que se congele la ligadura entre una variable y su valor una vez establecida dicha ligadura. La entidad resultante es, desde todos los puntos de vista, una constante simbólica definida por el usuario. Por ejemplo, en Pascal podemos escribir

`const pi = 3.1416`

y en ALGOL 68

`real pi = 3.1416`

y utilizar π en expresiones del tipo

`circunferencia := 2*pi*radio`

La variable π está ligada al valor 3.1416 y no se puede cambiar su valor, es decir, el traductor dirá que se ha producido un error si encuentra alguna asignación a π .

Pascal y ALGOL 68 se diferencian en el momento de efectuar la ligadura entre la variable y su valor inmodificable. En Pascal el valor puede ser un número o una cadena de caracteres y por lo tanto es posible establecer la ligadura durante el tiempo de traducción. El traductor puede sustituir en el programa su nombre simbólico por el valor de la constante. En ALGOL 68 el valor de la constante puede ser el producido por una expresión con otras variables y constantes. En consecuencia, la ligadura sólo se puede establecer en tiempo de ejecución, cuando las variables están creadas. Una constante manifiesta es una constante simbólica cuyo valor puede ligarse durante el tiempo de traducción.

Se puede plantear una pregunta relativa a la ligadura entre una variable y su valor: ¿Cuál es el valor de la variable inmediatamente después de su creación? Se pueden dar diferentes enfoques a esta cuestión. Desgraciadamente, la solución adoptada suele quedar sin especificar en la definición del lenguaje y se resuelve de modo distinto según las diferentes implementaciones. Este detalle hace bastante difícil el poder probar que el programa sea correcto, dado que esta corrección puede depender de la implementación. Ocurre además que el transportar un programa aparentemente correcto a una instalación diferente puede originar errores imprevistos o resultados inesperados.

La solución más obvia y más frecuentemente utilizada es ignorar el problema citado. En este caso, la cadena de bits que se encuentra en el área de memoria asociada a la variable se toma como su valor inicial. Otra solución posible sería proporcionar una estrategia de inicialización definida por el sistema; por ejemplo, los enteros se inicializan a cero, los caracteres a blancos, etc. Otra solución consiste en considerar una variable no inicializada como si estuviera inicializada a un

valor_sin_inicializar especial, y prohibir cualquier acceso de lectura a estas variables hasta que se les asigne un valor significativo. Esta solución es claramente la mejor y podría mejorarse en muchos aspectos. El único inconveniente sería el coste asociado a las comprobaciones necesarias efectuadas durante el tiempo de ejecución para asegurar que nunca se utilice en el programa un valor no inicializado.

3.3.4 Tipo de una Variable

Se puede definir el tipo de una variable como una especificación de la clase de valores que se pueden asociar a la variable, junto con las operaciones que se pueden usar para crear, acceder y modificar tales valores.

Cuando se define un lenguaje, generalmente se liga un nombre de tipo a cierta clase de valores y a un conjunto de operaciones. Por ejemplo, el tipo booleano se liga a los valores true y false, y a las operaciones and, or y not. Valores y operaciones se ligan a una cierta representación de máquina cuando se implementa el lenguaje. Por ejemplo, true puede ir ligado a la serie de bits 00...01 y false a la serie 00...00. Las operaciones and, or y not se pueden implementar mediante las adecuadas instrucciones de máquina que operan con las series de bits que representan valores lógicos.

Hay lenguajes en los que el programador puede definir nuevos tipos mediante una declaración de tipo. Por ejemplo, en Pascal se puede escribir

```
type t = array[1..10] of boolean
```

Esta declaración establece una ligadura (durante el tiempo de traducción) entre el tipo de nombre t y su implementación, es decir, una matriz de diez valores booleanos cada uno de ellos accesible por medio de un índice en el subrango 1 a 10. Como consecuencia de esta ligadura, el tipo t hereda todas las operaciones de la representación de la estructura de datos (la matriz) y por lo tanto se podrá leer y modificar cada componente de un objeto de tipo t indexándolo en la matriz.

En los lenguajes que soportan la definición de tipos abstractos de datos no hay una ligadura por defecto entre un tipo nuevo y su conjunto de operaciones. Las operaciones han de especificarse como un conjunto de subprogramas en la declaración del nuevo tipo. La declaración del nuevo tipo se hace de la forma siguiente:

```
type t = estructura de datos que representa objetos de tipo t;
        rutinas que se utilizan para manipular los objetos
        de tipo t
end
```

Los tipos pueden ir ligados a las variables de una manera

estática o dinámica. La solución estática es la adoptada por la mayoría de los lenguajes que iremos viendo, como Pascal, ALGOL 68, SIMULA 67, CLU, Ada, etc., y también por algunos más antiguos, como FORTRAN, COBOL, ALGOL 60 y PL/I.

En estos lenguajes, la ligadura entre la variable y su tipo se especifica generalmente mediante una declaración de variable. Por ejemplo, podemos escribir en Pascal

```
var x,y: integer;
    z: boolean;
```

Sin embargo, en algunos lenguajes (como FORTRAN), la primera aparición de un nuevo nombre de variable se toma como una declaración implícita. La ventaja de las declaraciones explícitas reside en la claridad de los programas y en su mayor fiabilidad, dado que durante el tiempo de traducción se pueden detectar detalles tales como errores de escritura. Por ejemplo, en FORTRAN, la declaración de la variable ALFA seguida de una sentencia tal como ALGA = 7.3, pretende en principio asignarle un valor y no sería detectado como error. "ALGA" no se consideraría como la aparición incorrecta de una variable no declarada, es decir, como ALFA mal escrito, sino que se tomaría como la declaración implícita de la nueva variable ALGA.

APL y SNOBOL4 utilizan ligadura dinámica entre variables y tipos. Por ejemplo, en APL un nombre de variable puede significar, en diferentes puntos de la ejecución del programa, bien una variable simple, una matriz unidimensional o multidimensional, e incluso una etiqueta. Las variables no se declaran explícitamente en APL, sino que su tipo queda determinado implícitamente por el valor que en ese momento contengan. Por ejemplo, después de ejecutar la siguiente sentencia de asignación,

```
A <-- 5
```

A es una variable entera que contiene el valor "5". Una sentencia posterior,

```
--> A
```

trataría A como una etiqueta y saltaría a la sentencia cuyo número fuera el valor de A. Posteriormente, A podría ser modificada con la siguiente sentencia:

```
A <-- 1 2 5 1 0
```

Ahora, A denota una matriz unidimensional de longitud 4. El límite inferior del índice quedaría establecido implícitamente a 1.

La ligadura dinámica supone una gran flexibilidad a la hora de crear y manipular estructuras de datos. Sin embargo, también presenta problemas en términos de disciplina de programación,

corrección del programa y eficacia de la implementación. Los programas son difíciles de leer, dado que el tipo de una variable que aparece en una sentencia no es conocido inmediatamente, sino que depende de los caminos de ejecución que siga el programa. De tal manera que una sentencia APL tal como:

$A[2;3] <-- 0$

pretende asignar el valor 0 al componente de la fila 2, columna 3, en la matriz bidimensional, y sólo será correcta si en un cierto punto de la ejecución A resulta ser una matriz bidimensional. Por ejemplo, sería incorrecto si $A <-- 0$ fuera la última asignación de A. En otras palabras, APL necesita de una comprobación de tipos dinámica para poder verificar que el uso que se hace de cada variable es consistente con su tipo. Por el contrario, la ligadura estática es la base de la comprobación de tipo estático y sus ventajas con respecto a la corrección del programa ya fueron expuestas en la Sección 2.4.

Pongamos otro ejemplo. Consideremos la sentencia APL

$A <-- B+C$

Esta sentencia es correcta siempre que cualquiera o ambas variables B y C sean variables simples, pero también en el caso de que B y C sean matrices con el mismo número y tamaño de dimensiones. Además, las acciones necesarias para ejecutar la sentencia dependen de los tipos de B y C. Si resultan ser variables simples, las acciones implicadas serían una simple suma. Si fueran matrices unidimensionales, las acciones serían un bucle de sumas y asignaciones.

Este ejemplo muestra que la información sobre tipos de variables APL debe utilizarse durante el tiempo de ejecución, no sólo para hacer la comprobación dinámica de tipos, sino también para elegir las acciones apropiadas para la ejecución de las sentencias. Con el fin de poder utilizar la información de tipos, deben existir descriptores durante el tiempo de ejecución que deberán ser modificados cada vez que se establezca una nueva ligadura. Por el contrario, otros lenguajes (tales como el Pascal) están diseñados de manera que la información de tipos se conoce durante el tiempo de traducción. De todo esto se deduce que los descriptores solamente son necesarios durante el tiempo de traducción.

Los lenguajes que adoptan la ligadura dinámica entre variables y tipos se procesan de una manera más natural mediante interpretación. Como hemos visto en el ejemplo anterior, generalmente no existe suficiente información, antes del tiempo de ejecución, para generar el código para la evaluación de expresiones que contengan variables de tipo desconocido. La elección entre traducción e interpretación en la implementación de un lenguaje resulta por lo tanto muy influenciada por la ligadura entre variables y tipos. Los lenguajes con ligadura dinámica suelen estar orientados a la interpretación, mientras

que los de ligadura estática lo están a la traducción.

3.4 UNIDADES DE PROGRAMA

Los lenguajes de programación hacen posible que un programa se componga de un cierto número de unidades. Estas unidades de programa se pueden desarrollar de una manera más o menos independiente, e incluso a veces traducirse por separado y combinarse después de la traducción. Las variables que se declaran dentro de una unidad se dice que son locales a la misma. Una unidad se puede activar durante la ejecución. Como ejemplos bien conocidos de unidades de programa tenemos los subprogramas en lenguaje ensamblador, subrutinas FORTRAN y procedimientos y bloques de ALGOL 60. En esta sección estudiaremos algunos mecanismos elementales que controlan el flujo de la ejecución entre unidades de programa y las ligaduras que se establecen al activar una unidad. En el Capítulo 5 se verá con todo detalle el importante tema del paso de parámetros a subprogramas y mecanismos adicionales que controlan el flujo de la ejecución.

La representación de una unidad de programa durante la ejecución recibe el nombre de activación de unidad. Una activación de unidad se compone de un segmento de código y de un registro de activación. El segmento de código, cuyo contenido es fijo, contiene las instrucciones de la unidad. El contenido del registro de activación es variable. Este registro contiene toda la información necesaria para ejecutar la unidad, incluyendo entre otras cosas los objetos asociados con las variables locales de una unidad de activación concreta. La posición relativa de un objeto en el registro de activación recibe el nombre de desplazamiento. Para referenciar un objeto, el procesador puede usar la dirección inicial del registro de activación que contiene el objeto y el desplazamiento del mismo.

Una unidad no es, en ningún caso, una parte del programa completamente independiente ni autosuficiente. En caso de que sea un subprograma, se le puede activar mediante una llamada realizada por otra unidad, a la que se le devolverá el control después de la ejecución. Por lo tanto, el punto de retorno es una información (susceptible de cambio) que se debe preservar en el registro de activación cuando se hace la llamada al subprograma. Además, las unidades pueden hacer referencia a variables que no están declaradas como locales, siempre que esto esté permitido por las reglas de ámbito del lenguaje. Las variables no locales que pueden ser referenciadas por una unidad reciben el nombre de globales para esa unidad. (*)

(*) Algunos autores, (p.ej., Pratt 1975) hacen distinción entre los términos "no local" y "global". Este último se utiliza para determinar una variable no local perteneciente a un registro de activación que está activo durante toda la

El entorno de referencia de una activación de la unidad U consta de las variables locales de U, que están ligadas a objetos almacenados en el registro de activación de U (entorno local), y de las variables globales de U ligadas a objetos almacenados en los registros de activación de otras unidades (entorno global). Dos variables del entorno de referencia de una unidad que se refieran al mismo objeto de datos reciben el nombre de alias. La modificación de un objeto de datos ligado a una variable global recibe el nombre de efecto lateral.

Las unidades pueden activarse recursivamente, esto es, una unidad puede llamarse a sí misma bien directa, bien indirectamente a través de alguna otra unidad. En otras palabras, se puede producir una nueva activación de una unidad antes de que termine la activación anterior. Todas las activaciones de la misma unidad se componen del mismo segmento de código pero de diferentes registros de activación. Por tanto, al producirse la recursión, la ligadura entre un registro de activación y su segmento de código es necesariamente dinámica. Siempre que se activa una unidad, se debe establecer una ligadura entre un registro de activación y su segmento de código, con el fin de producir una nueva activación de unidad.

Algunos lenguajes, como FORTRAN, no mantienen activaciones recursivas y por tanto la ligadura entre el segmento de código y el registro de activación puede ser estática, y la creación (o inicialización) de objetos para las variables locales de una unidad puede establecerse previamente a la ejecución del programa. En las secciones siguientes trataremos este asunto con más detalle. El caso de FORTRAN aparece en la Sección 3.5; el ALGOL y similares se estudian en la Sección 3.6 y por último, los temas básicos de los lenguajes interpretativos, como APL, en la Sección 3.7.

3.5 ESTRUCTURA DEL FORTRAN

Un programa FORTRAN se compone de un conjunto de unidades: un programa principal y un conjunto (posiblemente vacío) de subprogramas (subrutinas y funciones). La cantidad de memoria necesaria para contener cada variable local es fija y conocida durante el tiempo de traducción y no puede cambiarse durante la ejecución de la unidad.

Cada unidad se compila por separado y se asocia a un registro de activación al que se le asigna memoria antes de la ejecución, o lo que es lo mismo, las variables se pueden crear antes de la ejecución y su tiempo de vida se extiende a lo largo de toda la ejecución del programa (variables estáticas). Sin embargo, el ámbito de una variable queda limitado a la unidad en la cual se ha declarado.

ejecución del programa. Nosotros emplearemos ambos términos indistintamente.

Las unidades pueden tener acceso a variables globales que hayan sido declaradas mediante sentencias COMMON. Estas variables se pueden considerar como pertenecientes a un registro de activación suministrado por el sistema y que es global a todas las unidades de programa. En la Figura 3.1 se muestra un esquema de un programa FORTRAN.

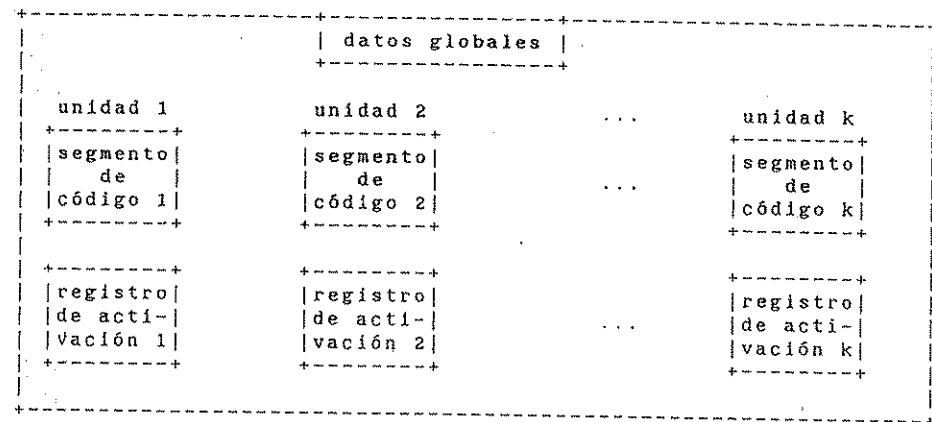


Figura 3.1 Programa FORTRAN durante su ejecución.

Cuando se traduce una unidad, a medida que se van procesando las declaraciones de variables, se reservan posiciones consecutivas del registro de activación para las variables locales de dicha unidad, es decir, los desplazamientos quedan estáticamente ligados a las variables. Sin embargo, la ligadura entre una variable y su área de almacenamiento no puede completarse, pues aún no se conoce el área donde se almacenará el registro de activación. Este conocimiento sólo se obtiene en el momento en que las unidades quedan montadas y cargadas antes de la ejecución.

Resumiendo, el FORTRAN permite un esquema de asignación estática de memoria. El almacenamiento requerido por cada unidad queda fijado y es conocido antes de la ejecución, puede asociarse a la unidad antes de la ejecución y permanece ligado a la unidad durante el transcurso de la ejecución del programa, aun cuando la unidad no esté activa. Una consecuencia de la asignación estática de memoria es que la mayoría de los procesadores FORTRAN permiten que las unidades muestren un comportamiento "sensible a la historia", a pesar de que esta característica está explícitamente prohibida en la definición estándar del lenguaje. Un subprograma sensible a la historia puede producir diferentes resultados cuando se activa dos veces con los mismos valores de parámetros y de variables globales, debido a que los valores almacenados en el entorno local pueden ser diferentes en cada activación. Por ejemplo, algunos valores locales se pueden inicializar antes de

la ejecución (mediante una sentencia DATA) y se actualizarán con cada activación de la unidad. Las unidades sensibles a la historia son casi siempre difíciles de comprender y analizar. Un subprograma no se puede describir mediante la simple enumeración de entradas y salidas, ya que su comportamiento puede depender del estado de los cálculos, es decir, de los valores almacenados en el registro de activación.

La asignación estática de memoria resulta fácil y simple de implementar, pero el lenguaje resultante carece de flexibilidad. No es posible escribir subprogramas recursivos. Todas las variables, sin excepción (en particular las matrices) deben tener un tamaño fijo y estáticamente conocido. Por último, la memoria queda ocupada por los registros de activación aun cuando no estén activas las correspondientes unidades.

3.6 ESTRUCTURA DE LENGUAJES TIPO ALGOL

La mayoría de los lenguajes que veremos en los capítulos siguientes son descendientes del ALGOL 60, y se les suele llamar lenguajes tipo ALGOL. Estos lenguajes poseen una característica, la estructura de bloques, que sirve para controlar el ámbito de las variables y para dividir los programas en unidades. Dos unidades cualesquiera en el texto del programa pueden ser bien disjuntas, es decir, no tienen ninguna parte común, o bien anidadas, es decir, una de las unidades engloba totalmente a la otra. Por lo tanto, la estructura del programa se puede considerar como un anidamiento estático de unidades (véase Figura 3.2 (a)). Este anidamiento estático se puede representar también mediante una estructura de árbol. La estructura de árbol que se muestra en la Figura 3.2 (b) para el programa de la Figura 3.2 (a), muestra claramente que las unidades B y E están encerradas estáticamente dentro de la unidad A, las unidades F y G están a su vez encerradas estáticamente dentro de E, C lo está en B, y D en C.

Si se declara una variable local a una unidad U, resulta que es visible en U pero no en las unidades que contienen a U. Sin embargo, si es visible (con una excepción), para todas las unidades que están estáticamente anidadas dentro de U. En la Figura 3.2, una variable declarada en la unidad B no es visible para A, E, F ni G; pero sí lo es para B, C y D. Es local a B y global a C y D.

La excepción a la regla citada anteriormente se produce cuando a una variable local a una unidad determinada, se le da el mismo nombre que a una variable declarada en otra unidad que engloba a la primera. En tal caso, el mismo nombre puede significar o bien el objeto declarado localmente, o el objeto global declarado en la unidad más externa. El convenio para los lenguajes tipo ALGOL es que las declaraciones locales enmascaran las declaraciones globales. Esto implica que en la Figura 3.2, si declaramos una variable v en A y en C, ocurrirá que cualquier referencia a v desde A, B, E, F o G, se referirá a la variable v.

local a A; mientras que cualquier referencia a v dentro de C y D se refiere a la variable v local a C.

En general, para determinar qué variables son visibles a una unidad, es necesario proceder desde esa unidad hacia afuera a través de todos los niveles que encierran el anidamiento estático. Todas las declaraciones de variable que no fueran encontradas previamente, definen un nombre visible para la unidad.

```
/unidad A
    /unidad B
        /unidad C
            /unidad D
                - 
                \end D
            - 
            \end C
        - 
        \end B
    - 
    /unidad E
        /unidad F
            - 
            \end F
        /unidad G
            - 
            \end G
        - 
        \end E
    - 
\end A
```

(a)

(b)

Figura 3.2 Anidamiento estático de unidades tipo ALGOL.
 (a). Disposición de las unidades. (b) Árbol de anidamiento estático.

En los lenguajes tipo ALGOL, las unidades pueden ser de dos tipos: bloques y subprogramas. Los bloques se activan cuando aparecen durante la progresión normal de la ejecución y solamente sirven para definir un nuevo entorno mediante declaraciones locales. Los subprogramas son unidades con nombre que sólo se activan cuando se las llama explícitamente. Las reglas de ámbito de los nombres de subprograma son las mismas que las ya descritas para variables.

Con objeto de modelar el comportamiento en ejecución de los lenguajes tipo ALGOL, es necesario especificar las normas que gobiernan el tiempo de vida de las variables y la creación de los entornos de referencia para la activación de unidades. Generalmente, todas las variables locales de una unidad se crean automáticamente cuando tal unidad se activa, de manera que podemos afirmar que no hay una operación de creación explícita. Algunos lenguajes requieren que la cantidad de memoria necesaria para cada variable sea fija y conocida estáticamente; este caso lo expondremos en la Sección 3.6.1. En otros lenguajes, la cantidad de almacenamiento necesario para cada variable sólo se conoce, en la mayoría de los casos, durante la ejecución, al activarse la unidad; este caso se verá en la Sección 3.6.2. En otros casos, el programador crea explícitamente las variables locales. Consecuentemente, la cantidad de almacenamiento requerida por un registro de activación ni siquiera se conoce al activarse la unidad, sino que crece dinámicamente al ejecutarse nuevas sentencias de creación, como veremos en la Sección 3.6.3.

En la Sección 3.6.4 podremos ver un importante aspecto final, y es cómo acceden a su entorno global las activaciones de unidades de tipo ALGOL.

3.6.1 Registro de Activación de Tamaño Conocido Estáticamente

En este caso, las variables locales se crean implícitamente al activarse la unidad, y la cantidad de almacenamiento necesario para contener el valor de cada variable local se conoce en tiempo de traducción. Pascal (si hacemos caso omiso de los punteros) y C son dos lenguajes de este tipo.

Al igual que en el caso de FORTRAN, el tamaño del registro de activación y el desplazamiento de cada variable local se conocen en el tiempo de traducción. Sin embargo, el registro de activación no puede ir ligado estáticamente al segmento de código de la unidad antes de la ejecución, puesto que pueden producirse varias activaciones recursivas de la unidad al mismo tiempo. Por tanto, se reduce que el registro de activación debe ser asignado y ligado dinámicamente para cada nueva activación. Consecuentemente, durante la traducción, una variable solamente puede estar ligada a su desplazamiento dentro del registro de activación. La ligadura a su dirección física de memoria requiere conocer la dirección del registro de activación y sólo puede efectuarse durante la ejecución. Las variables de este tipo reciben el nombre de variables semiestáticas.

Supongamos por ejemplo que aparece la siguiente declaración en una unidad:

`a: array[0..10] of integer`

es decir, a es una matriz de enteros con subíndice en el rango de 0 a 10. Supongamos también que cada entero ocupa una posición en

el registro de activación y que están asignadas a la matriz posiciones consecutivas de memoria. Durante la traducción se conoce la dirección de a relativa a la dirección de la primera posición del registro de activación, que se conocerá en la ejecución. Si x es una variable que contiene esta dirección, una referencia a `a[i]` se traducirá por una referencia a la posición `x+desplazamiento_de_ati`.

La asignación dinámica de los registros de activación produce dos efectos fundamentales: por una parte permite la implementación de activaciones recursivas de unidad, y por otra supone una utilización más eficaz de la memoria central, tal y como veremos más adelante. Con el fin de que sea posible hacer el retorno desde una activación, es necesario que los registros de activación contengan suficiente información para identificar la instrucción que se ha de ejecutar a continuación, así como para qué se active (con el retorno) el anterior registro de activación. La primera información (dirección de retorno) está formada por dos informaciones: un puntero al segmento de código de la unidad llamante y un desplazamiento dentro de ese segmento. La segunda está representada por un puntero al registro de activación de la unidad llamante. Este puntero se denomina enlace dinámico. La cadena de enlaces dinámicos que se originan en el registro de activación en curso se denomina cadena dinámica. Esta cadena representa el anidamiento dinámico de las activaciones de unidades.

Por ejemplo, si las unidades F y G del programa de la Figura 3.2 son subprogramas mutuamente recursivos, la Figura 3.3 representa una descripción parcial del estado de nuestro procesador abstracto después de producirse la siguiente cadena de llamadas:

- Call E realizado por la instrucción 3 dentro del segmento de código de A.
- Call F realizado por la instrucción 5 dentro del segmento de código de E.
- Call G realizado por la instrucción 7 dentro del segmento de código de F.
- Call F realizado por la instrucción 3 dentro del segmento de código de G.
- Call G realizado por la instrucción 7 dentro del segmento de código de F.
- Call F realizado por la instrucción 3 dentro del segmento de código de G.

Cuando una unidad U termina su activación, ya no es necesario el registro de activación. Las normas del tiempo de vida especifican que cada activación de U debe tener un nuevo registro de activación. Las variables locales a U sólo pueden ser globales a las unidades anidadas en U y que a su vez estén activadas tras la presente activación de U. De ahí que tales activaciones se completen antes que la activación de U en curso. Por tanto, cuando una unidad ha completado su activación en curso, es posible liberar el espacio ocupado por el registro de

activación para que quede disponible para situar nuevos registros de activación. Puesto que el registro de activación que se libera es el que se asignó más recientemente, los registros de activación se pueden asignar siguiendo una política LIFO en un almacenamiento organizado como pila.

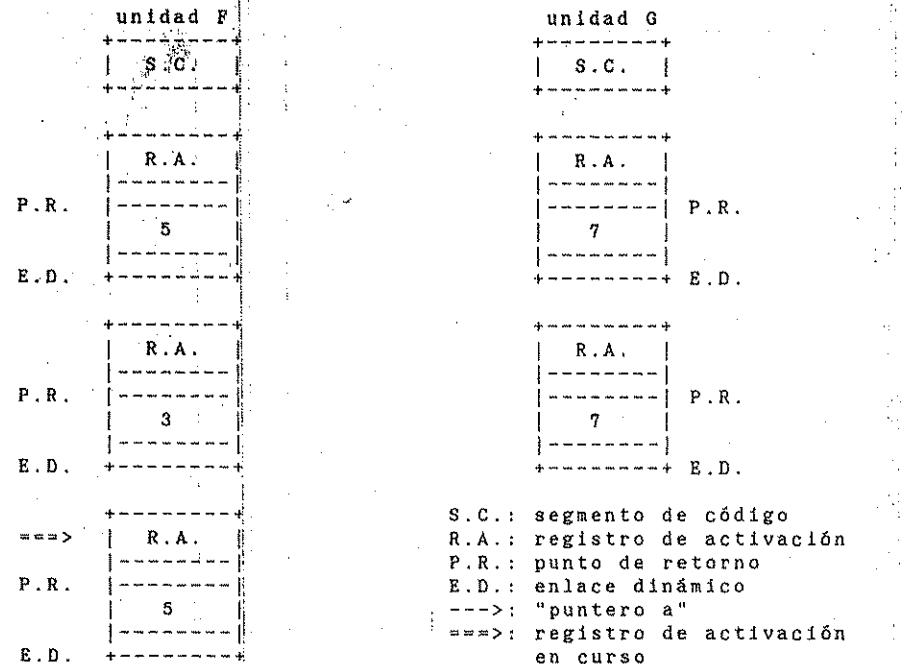


Figura 3.3 Representación parcial del estado del procesador abstracto durante la ejecución del programa de la Figura 3.2.

La implementación en la gestión de la pila se puede hacer de la manera siguiente: Sean LIBRE y EN_CURSO dos variables que apuntan a la primera posición libre de la pila y a la primera posición del registro de activación que ocupe el lugar más alto (el que está activo en ese momento precisamente) respectivamente (véase Figura 3.4).

Las acciones que se llevan a cabo cuando se activa una unidad son las siguientes:

1. PILA[LIBRE] := EN_CURSO (establecer el enlace dinámico para el nuevo registro de activación)
2. EN_CURSO := LIBRE (poner el nuevo valor de EN_CURSO)

3. LIBRE := LIBRE+T (T es el tamaño del nuevo registro de activación)

Cuando la unidad termina la ejecución, se puede hacer un seguimiento del enlace dinámico para que llegue al registro de activación de la unidad llamante. Las acciones que se precisan son las siguientes:

1. LIBRE := EN_CURSO

2. EN_CURSO := PILA[EN_CURSO]

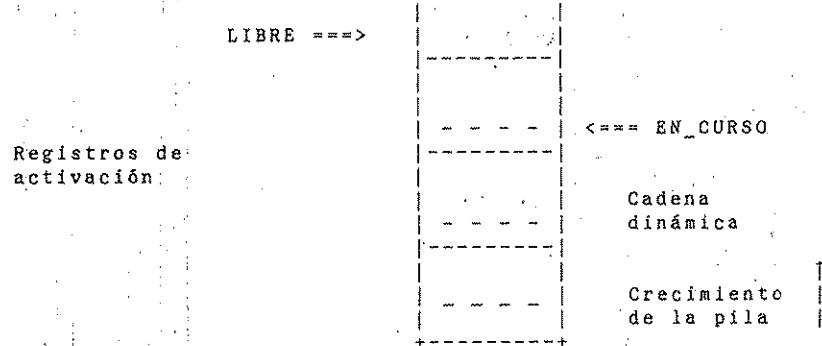


Figura 3.4 Enlaces y cadena dinámica de la pila de ejecución.

La gestión de la memoria organizada en pila para los lenguajes tipo ALGOL es una elección de implementación y no está estrictamente condicionada por la semántica del lenguaje. En realidad, la semántica sólo exige que las variables locales estén ligadas al nuevo registro de activación para cada nueva activación, y conceptualmente, los objetos locales de las activaciones previas pueden continuar existiendo indefinidamente. Sin embargo, y gracias a las reglas de ámbito del lenguaje, un registro de activación se hace inaccesible tan pronto como termina la activación. Esta es la razón por la cual podemos optar por situar los registros de activación en una pila. La capacidad para reutilizar el almacenamiento liberado por los registros de activación es la razón por la cual la asignación dinámica de almacenamiento resulta en principio más eficaz que la asignación estática.

3.6.2 Registros de Activación cuyo Tamaño es Conocido en la Activación de la Unidad

En ALGOL 60 y otros lenguajes de la misma familia, como ALGOL 68 y más recientemente Ada, nunca se conocen estáticamente

ni el tamaño del registro de activación ni la posición de cada variable local dentro del registro de activación. Las variables se crean automáticamente al activarse la unidad, pero su tamaño puede depender de valores que sólo se conocerán en el momento de la ejecución al activarse la unidad. Tal es el caso de las matrices dinámicas, cuyas ligaduras se dan a conocer durante la ejecución. A las variables que respondan a estas características las denominaremos variables semidinámicas.

La introducción de variables semidinámicas añade ciertas complicaciones al caso expuesto en la Sección 3.6.1, tales como que las variables locales no pueden estar ligadas a un desplazamiento constante dentro del registro de activación durante la traducción y deberá retrasarse la ligadura de dirección hasta la ejecución. Supongamos, por ejemplo, que la unidad U aparece en las siguientes declaraciones de ALGOL 68:

```
[1:n] int a;
[1:m] real b;
```

es decir, a es una matriz de enteros con índice en el rango de 1 a n, y b es una matriz de reales con índice de rango de 1 a m. Sean m y n dos variables globales. Dado que los valores de m y n no son conocidos en tiempo de traducción, será imposible determinar estáticamente la cantidad de memoria necesaria para a y b, y por consiguiente, para el registro de activación de U. Sin embargo, la semántica del lenguaje requiere que los valores de m y n (y por tanto, el tamaño del registro de activación) sean conocidos en el momento de la activación de la unidad. Todo esto es de suma importancia para llevar a cabo una implementación eficaz, tal y como veremos a continuación.

Durante la traducción podemos reservar, en el registro de activación, espacio para los descriptores de las matrices dinámicas a y b. El descriptor incluye al menos una posición en la que se almacena un puntero para el área de almacenamiento de la matriz dinámica y otra para los límites inferior y superior de cada dimensión de la matriz. Dado que durante la traducción ya se conoce el número de dimensiones de la matriz, se conocerá estáticamente el tamaño del descriptor. Todos los accesos a variables semidinámicas se traducirán como referencias indirectas a través del puntero en el descriptor, cuyo desplazamiento se determina estáticamente. La asignación del registro de activación pasa por varias etapas durante la ejecución. En primer lugar se asigna el espacio necesario para las variables semiestáticas y para los descriptores de las variables semidinámicas. Cuando aparece la declaración de una variable semidinámica, se establecen los valores de las entradas de dimensión en los descriptores, se evalúa el tamaño real de la variable semidinámica y se expande el registro de activación para incluir la variable. (Esta expansión es posible debido a que, estando activa la unidad, el registro de activación está en la parte superior de la pila). Por último, el puntero en el descriptor se coloca apuntando al área que se acaba de asignar.

3.6.3 Registros de Activación de Tamaño Variable Dinámicamente

Los lenguajes descritos en las secciones anteriores se caracterizan por tener implícitamente creadas todas las variables locales en el momento de la activación de la unidad. Además, el tamaño de los registros de activación o bien se conoce estáticamente, o en el peor de los casos se conocerá cuando se activen las unidades. Los lenguajes de la familia del ALGOL (excepto el ALGOL 60) desde PL/I y Pascal hasta ALGOL 68 y Ada, permiten a los programadores manejar objetos de datos cuyos tamaños sean variables durante la ejecución. Por consiguiente, no se conocerá al activarse la unidad la cantidad de memoria requerida por un registro de activación. Estas variables reciben el nombre de variables dinámicas.

Como ejemplo notable de variable dinámica podemos citar la matriz flexible del ALGOL 68 y otros lenguajes. Una matriz flexible es aquella cuyos límites de las dimensiones pueden variar a lo largo de la ejecución del programa para acomodar el tamaño del objeto que le haya sido asignado. Supongamos la siguiente estructura de un programa tipo ALGOL:

```
begin
  Unit A;
  begin
    Unit B;
    end B;
  end A;
```

Siendo x una matriz flexible declarada como local en la unidad A, supongamos una asignación a x dentro de la unidad B. El espacio para contener el valor de x no se puede reservar en la pila del registro de activación de A, debido a que sólo conoceremos la cantidad de espacio necesario para contener el valor asignado por la unidad B, cuando B se ejecute, es decir, cuando el registro de activación de B ocupe la cima de la pila de ejecución. En ese punto de la ejecución, el registro de activación de A ocupa una posición más baja en la pila, y al cambiar su tamaño para acomodar el objeto recién asignado a x, requeriría una remodelación de la pila de ejecución, solución que sería evidentemente irrealizable.

Otro ejemplo de variables dinámicas son aquellas que se asignan bajo control del programa. Estas variables existen en PL/I, Pascal y otros lenguajes, y permiten la creación de estructuras de datos susceptibles de expandirse y contraerse. Tales estructuras de datos pueden modelarse como si consistieran en un conjunto de nodos. Estos nodos pueden ser añadidos y borrados de la estructura dinámicamente. Como ejemplo de estas estructuras de datos podemos citar las listas encadenadas y los árboles. Generalmente, los nodos van conectados mediante punteros. Dado que a los nodos se les asigna memoria durante la ejecución, y que su número no se conoce ni en el momento de escribir el programa ni en la traducción, es imposible nombrarlos

explicitamente. El acceso se hace indirectamente mediante punteros. Por ejemplo, en Pascal cada puntero está calificado de tal manera que sólo puede apuntar a objetos de un determinado tipo. Supongamos que *p* ha sido declarado como puntero de tipo *t*, entonces la sentencia

new(p)

crearía un objeto de tipo *t* y asignaría su dirección a *p*. La vida de un objeto creado de este modo, a diferencia de las variables semiestáticas y semidinámicas, no termina cuando se sale del bloque que contiene la sentencia de asignación, sino que por el contrario, algunos lenguajes (p.ej., PL/I) tienen sentencias para desasignar explícitamente estos objetos. Otros lenguajes (como Pascal) afirman que el objeto vive mientras dure una referencia a él, es decir, un puntero. Además, es posible crear varios de estos objetos sin desasignar ninguno de ellos. Por tanto, es fácil comprender que a estos objetos no se les puede asignar memoria en la pila.

Consideremos de nuevo la estructura de programa citada anteriormente: sea *p* un puntero declarado en la unidad A, y supongamos que B contiene la sentencia de asignación "*new(p)*". Cuando se ejecute la sentencia de asignación de memoria, el registro de activación que esté en la cima de la pila será el de la unidad B. El objeto no puede ir asignado a la parte superior de la pila porque cuando se sale de B el objeto quedaría desasignado (mientras que *p* todavía apunta a él). Tampoco es posible asignar al registro de activación de A, ya que sería necesario remodelar la pila de ejecución como ya vimos en el caso de las matrices flexibles.

En resumen, las variables dinámicas denotan objetos cuyo tamaño y/o número puede variar dinámicamente a lo largo de su vida. Este hecho imposibilita la asignación de estas variables a una pila, y hace que se asignen a un área de memoria denominada libre. El término memoria libre indica que se libera de la connotación de la pila LIFO. Por esta razón, las variables dinámicas reciben el nombre de variables de memoria libre, en oposición a las variables semiestáticas y semidinámicas, que reciben el nombre de variables de pila. En las Secciones 4.7.2.6 y 4.7.4 se detallan más a fondo las variables de memoria libre y la gestión de la memoria libre.

3.6.4. Acceso al Entorno Global

Hasta ahora, solamente hemos visto cómo la activación de una unidad puede referenciar su propio entorno local. En la presente sección veremos cómo las activaciones de unidades pueden hacer referencia a su entorno global. Dejaremos el problema de cómo pasar parámetros para el Capítulo 5.

Como ya hemos visto, en un lenguaje como FORTRAN se considera a las variables globales como pertenecientes a un

registro de activación proporcionado por el sistema, que es global a todas las unidades, y todas las ligaduras entre variables globales de una unidad y el almacenamiento asignado a ellas puede establecerse estáticamente.

En los lenguajes tipo ALGOL, aquellos registros que estén en la pila en un momento determinado representan la cadena dinámica de activaciones de unidades. La Figura 3.5 muestra la pila de nuestro procesador abstracto tras la secuencia de llamadas ya descrita en la Sección 3.6.1 para el programa de la Figura 3.2.

Supongamos que la variable entera *x* se declara en las unidades E y G, y que la variable entera *y* se declara en G, B y A. Además, supongamos que *z* es una variable entera local a F. Supongamos que el procesador abstracto llega a la sentencia *z:=x+y* cuando la pila de ejecución es la que se muestra en la Figura 3.5. Tal y como hemos visto, la ligadura apropiada para *z* a una posición de la pila se establece parcialmente en la traducción, convirtiendo cada aparición de *z* en *EN_CURSO+desplazamiento_de_z* (desplazamiento_de_z es un valor conocido estáticamente) y se completa durante la ejecución, al conocerse el valor de *EN_CURSO* (la dirección inicial del registro de activación de F). Pero, ¿qué ocurre con *x* e *y*? Obsérvese que la ligadura entre *x* (o *y*) y la posición de la pila a ella asignada no es la ligadura establecida más recientemente. De hecho, la última ligadura a una posición de pila para la variable *x* ha quedado establecida por la última activación de la unidad G, pero las reglas de ámbito del lenguaje requieren que la variable *x* referenciada en F sea la que se declare en E. Del mismo modo, la variable "y" referenciada en F, es la que se ha declarado en A y no la que se ha asignado más recientemente por la última activación de G.

En otras palabras, la secuencia de los registros de activación almacenados en la pila representa la secuencia de las activaciones de unidad generadas dinámicamente durante la ejecución. Lo que determina el entorno no local son las reglas de ámbito del lenguaje basadas en el anidamiento estático de las declaraciones de unidad.

3.6.4.1 Cadena Estática

Una manera de permitir el acceso a las variables globales es mantener un puntero (enlace estático) en la zona baja de la pila para cada registro de activación. Dicho puntero señalará al registro de activación de la unidad en la que esté englobado estáticamente según el texto del programa.

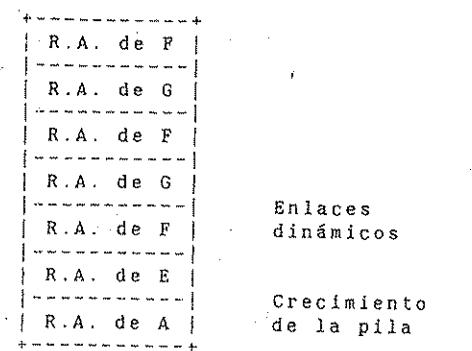


Figura 3.5 Pila del procesador para el caso de la Figura 3.3.

La Figura 3.6 nos muestra los enlaces estáticos para el ejemplo de la Figura 3.5. La secuencia de enlaces estáticos que se puede seguir desde el registro de activación más alto recibe el nombre de cadena estática. La referencia a variables globales puede explicarse intuitivamente como una búsqueda a través de la cadena estática. Para encontrar la ligadura correcta entre una variable y una posición de la pila, buscamos hacia abajo en la cadena estática hasta encontrar una ligadura. En nuestro ejemplo, la referencia a *x* se liga a una posición de la pila dentro del registro de activación de *E*, mientras que una referencia a *y* se liga a una posición de la pila dentro del registro de activación de *A*.

La búsqueda, que requeriría un elevado tiempo de ejecución, nubanca es necesaria en la práctica. Hay una solución más eficaz y está basada en el hecho de que el registro de activación que contiene una variable nombrada en una unidad *U*, se encuentra siempre a una distancia fija del registro de activación de *U*, a lo largo de la cadena estática. Si es una variable local, la distancia es, evidentemente, cero; si es una variable declarada en la unidad inmediatamente más externa, la distancia sería uno, y así sucesivamente. Este atributo de distancia se puede evaluar y ligar a la variable durante el tiempo de traducción. Por lo tanto, las variables pueden estar ligadas estáticamente a un par ordenado (distancia, desplazamiento) dentro del registro de activación. Si la variable es semiestática, el desplazamiento será la posición relativa del objeto dentro del registro de activación. Si la variable es semidinámica, el desplazamiento será la posición relativa de un puntero que apunta a la posición de la pila donde se encuentra el objeto. Si la variable es dinámica, el desplazamiento es la posición relativa de un puntero que apunta al área libre donde está ubicado el objeto.

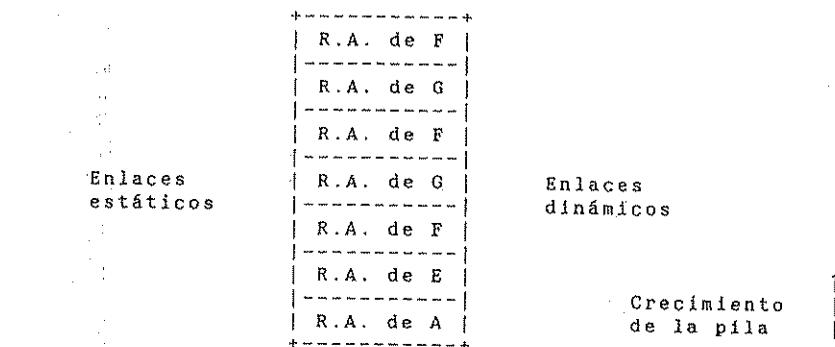


Figura 3.6 Ejemplo de la Figura 3.5 con enlaces estáticos.

Durante la ejecución, el par (distancia, desplazamiento) se utiliza de la siguiente forma: si *d* es el valor de la distancia, comenzando a partir de la posición EN_CURSO, avanzamos *d* pasos a lo largo de la cadena estática. Entonces, el valor del desplazamiento se añade a la dirección que se ha encontrado, siendo el resultado la dirección real del objeto global en tiempo de ejecución. En el caso de las variables semidinámicas y dinámicas, este proceso nos conduce a una posición a partir de la cual se precisa de un nivel extra de direccionamiento indirecto para alcanzar el objeto.

Hemos visto hasta el momento cómo utilizar la cadena estática para acceder a variables globales. Ahora veremos cómo se puede establecer el enlace estático durante la ejecución, cuando se activa una unidad. Si la unidad *U* es un bloque, entonces será muy fácil establecer la ligadura estática para una activación de *U*, ya que es evidentemente la misma que en el enlace dinámico. Si *U* es un subprograma, la operación resultante es más compleja. En primer lugar, para cada subprograma local declarado en una unidad se utiliza una entrada en el registro de activación de dicha unidad para almacenar un puntero al segmento de código del subprograma. En nuestro procesador abstracto podríamos entonces representar una llamada a un subprograma como "salto(dist,desp)" (*), donde *dist* (para las variables) es el número de pasos que hay que descender en la cadena estática y *desp* es el desplazamiento dentro del registro de activación que contiene un puntero que señala al segmento de código apropiado.

(*) Estamos considerando subprogramas sin parámetros.

Supongamos que se está procesando la unidad V y que se ejecuta la instrucción "salto(2,4)" para llamar a un subprograma U declarado como local a la unidad W. El enlace estático para el registro de activación de U se establece de manera que apunte a la base del registro de activación que está situado dos pasos más abajo en la cadena estática (véase Figura 3.7).

Cuando termina la ejecución de la unidad que en ese momento está activa, el retorno al entorno previamente activo no requiere ningún mecanismo adicional. El registro de activación que está en la parte superior sencillamente se borra y la cadena estática necesaria para acceder a un nuevo entorno global se encuentra ya instalada.

El mayor inconveniente que presenta la utilización de enlaces estáticos es la necesidad de seguir la cadena estática para cada referencia a una variable no local. El tiempo empleado en localizar un objeto global no es constante y depende del número de pasos que se hayan descendido en la cadena estática.

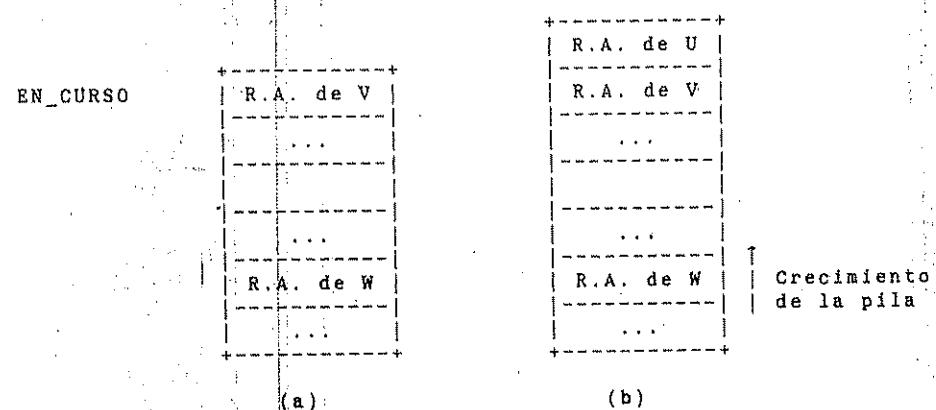


Figura 3.7 (a) Enlaces estáticos antes de llamar a U.
(b) Enlaces estáticos después de llamar a U.

3.6.4.2 Display

Una técnica alternativa de implementación es la utilización del "display" (original de Dijkstra), que acelera el proceso de referencia a objetos globales a base de hacer más compleja la activación de la unidad y su retorno. Esta nueva solución resulta útil si las referencias no locales aparecen con más frecuencia que las salidas y entradas a subprogramas.

En vez de utilizar enlaces estáticos, se emplea un vector de

longitud variable (el display) que contiene los punteros que señalan a los registros de activación en la cadena estática en cualquier punto dado durante la ejecución. La Figura 3.8 ilustra el display para el ejemplo de la Figura 3.6. A continuación detallamos los dos pasos necesarios para acceder a un identificador representado por el par (dist,desp):

1. Encontrar la dirección base b del registro de activación. Si m es el número de entradas actual en el vector DISPLAY (esto es, $0..m-1$ es el rango de los subíndices) entonces, $DISPLAY[m-dist-1]$ contiene el valor requerido de b.
2. Evaluar $b+desp$ para obtener la dirección de la variable.

Vemos por tanto que referenciar una variable global mediante un display resulta muy sencillo y requiere un tiempo de acceso constante para todas las variables globales. Sin embargo, resulta necesario implementar acciones que modifiquen el display en la entrada y salida de la unidad para mostrar o reflejar la cadena estática activa en ese momento. Una vez más, si la unidad es un bloque, las acciones a seguir son sencillas, y se las dejamos al lector para que le sirva de ejercicio. En el caso de una llamada a un subprograma, supongamos que P_0, P_1, \dots, P_{m-1} son los valores almacenados en DISPLAY con subíndices $0, 1, \dots, m-1$ antes de la llamada; supongamos también que la llamada que se va a procesar es "salto(dist,desp)". Si $j = m-dist-1$, entonces P_j ($0 \leq j < m$) es la referencia al registro de activación de la unidad donde el subprograma ha sido declarado como local. Luego el nuevo display contendrá $j+2$ entradas (las primeras $j+1$ se quedarán igual que antes de la llamada y la superior será una referencia al nuevo valor de EN_CURSO). Este caso, para el ejemplo de la Figura 3.7, se muestra en la Figura 3.9.

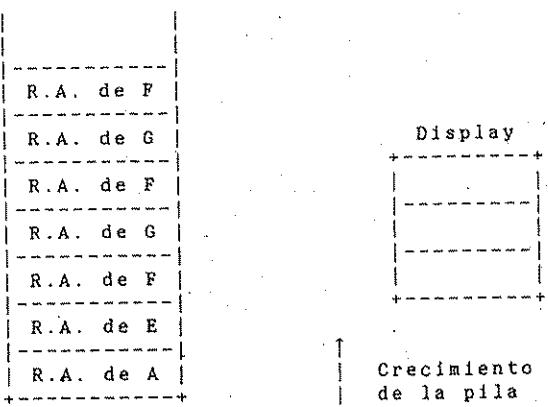


Figura 3.8 Solución con display para el ejemplo de la Figura 3.6.

El retorno desde un subprograma requiere que se restaure el contenido del display a lo que había antes de que se ejecutara la llamada. En el caso citado anteriormente, significa que los valores $P_{j+1}, P_{j+2}, \dots, P_{M-1}$ se deben haber conservado en el registro de activación del subprograma llamado. Una vez que se sale, estos valores se vuelven a copiar en el display.

3.7 ESTRUCTURA DE UN LENGUAJE INTERPRETATIVO: APL

En la Sección 3.3.1 decíamos que algunos lenguajes están orientados a la traducción y otros a la interpretación. De lo expuesto en la Sección 3.6 podemos deducir que los lenguajes tipo ALGOL están orientados a la traducción. La adopción de reglas de ligadura estática para ámbitos requiere que se haga la ligadura de una referencia de variable a una declaración de variable basándose en un examen de la estructura del programa. Y este examen lo realiza mejor un traductor que un intérprete. Por el contrario, la adopción de reglas de ámbito dinámico favorece el uso de un intérprete. A pesar de que este libro hace más énfasis en los lenguajes orientados a la traducción, el estudio de un lenguaje interpretativo, como APL, ayudaría a clarificar los puntos básicos y a contrastar las diferencias entre ambos enfoques.

APL tiene muchas características interesantes y poco corrientes. Desde ser interactivo y tener un cierto número de características especiales para utilizar en un entorno interactivo, hasta disponer de medios para facilitar la manipulación de matrices. Sin embargo, haremos caso omiso de la mayoría de estas características y sencillamente expondremos, en líneas generales, el comportamiento en ejecución de un procesador abstracto APL. En el Capítulo 8 se estudian los aspectos funcionales de APL.

Un programa APL consiste generalmente en un número de subprogramas y en una secuencia de sentencias las cuales podemos considerar como el programa principal (Figura 3.10). La primera línea de una definición de un subprograma declara los parámetros formales (I en el caso de SUB, N en el caso de FUN). Los subprogramas de tipo función también indican el parámetro de resultado (R en el caso de FUN). Las variables locales también se declaran (X en el caso de FUN, Y en el caso de SUB).

Las variables globales se declaran implícitamente mediante una asignación a un identificador que no ha sido declarado como local. La declaración de una variable no especifica el tipo. Los nombres 'de subprograma' se consideran como identificadores globales.

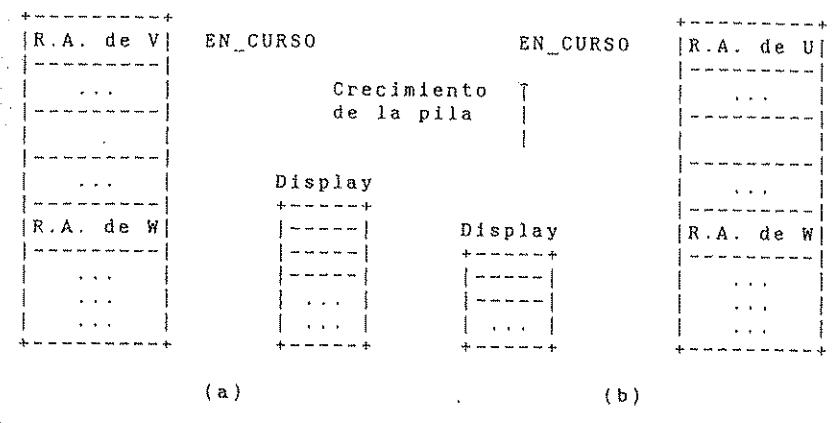


Figura 3.9 Implementación mediante display del ejemplo de la Figura 3.7.

- (a) Pila y display antes de llamar a U.
- (b) Pila y display después de llamar a U.

Las reglas de ámbito en APL son dinámicas, es decir, el ámbito de un nombre es totalmente dependiente de la cadena de llamadas en ejecución (la cadena dinámica), en lugar de serlo de la estructura estática del programa. En el ejemplo de la Figura 3.10, considérese el punto en el que se efectúa la llamada a SUB 2. Las referencias no locales a X y Z dentro de la activación de SUB se ligan a las X y Z globales, definidas en el programa principal. Cuando desde SUB se activa la función FUN, la referencia no local a Y se liga a la última definición de Y, es decir, al objeto asociado a Y en el registro de activación de SUB. La siguiente instrucción del programa principal llama de nuevo a la función FUN. En este caso, la referencia no local a Y desde FUN se liga a la Y global definida en el programa principal.

La implementación del mecanismo de referenciación global de APL puede ser muy sencilla. Los registros de activación se colocan en la pila de ejecución y se unen mediante enlaces dinámicos. Cada entrada del registro de activación registra explícitamente el nombre de la variable y contiene un puntero que señala al área de memoria libre en la que se puede almacenar el valor. La asignación a un área libre es necesaria porque la cantidad de memoria requerida por cada variable puede variar dinámicamente (véase Sección 3.3.1). Para cada variable (T por ejemplo), la búsqueda en la pila se hace siguiendo los enlaces dinámicos. La primera asociación que se encuentre para T en un registro de activación será la correcta. La Figura 3.11 nos muestra la pila de ejecución para el programa de la Figura 3.10 cuando SUB llama a FUN, que a su vez fue llamada por el programa principal.

```

PROGRAMA PRINCIPAL
Z <--- 0
X <--- 5
Y <--- 7
SUB 2
Z <--- FUN Y

```

```

SUBPROGRAMA SUB
...X...
...Y...
Z <--- FUN I

```

```

SUBPROGRAMA FUNCION FUN
...X...
...Y...
V R <--- FUN N; X

```

Figura 3.10 Estructura de un programa APL.

Este mecanismo de acceso, a pesar de ser muy simple, resulta muy poco eficaz. Siguiendo la misma filosofía que nos conduce a la idea del display para los lenguajes tipo ALGOL, es posible mantener aquí una tabla de referencias no locales activas en ese momento. En esta tabla sería suficiente con una sola búsqueda en lugar de tener que buscar en toda la cadena dinámica. No analizaremos esta solución con detalle. El lector observará que al igual que en el caso del display, esta solución acelera la referencia a variables no locales a costa de acciones más elaboradas, las cuales se ejecutarán a la salida y a la entrada del subprograma y actualizarán la tabla de referencias no locales activas.

Las reglas de ámbito dinámico de APL representan un gran avance en comparación con las reglas estáticas de los lenguajes tipo ALGOL. Antes de terminar esta sección, hagamos una breve evaluación del estilo de programación que soportan estas reglas de ámbito y de la fiabilidad de los programas resultantes. Las

reglas de ámbito dinámico ayudan a seguir la lógica del programa tal y como se ejecutará: la ligadura más reciente será la válida en cualquier instante. De aquí que las reglas de ámbito dinámico faciliten la ejecución interactiva de programas mediante un intérprete. Por el contrario, las reglas de ámbito estático ayudan a seguir la lógica del programa tal y como está escrito: el entorno de referencia en cualquier punto durante la ejecución de un programa no depende de los cálculos anteriores, sino de la estructura textual del programa. Por lo tanto, con las reglas de ámbito estático resulta más natural desarrollar un programa empezando primero por escribirlo, pasando luego a revisar el texto para comprobar su corrección, luego a traducirlo y, por último, a ejecutarlo.

La interpretación interactiva de programas es típica de los lenguajes dinámicos tales como APL, y esta solución resulta muy atractiva para programas de tamaño moderado. Como veremos en el Capítulo 8, el disponer de operaciones predefinidas muy potentes para las matrices, hace que el lenguaje sea particularmente adecuado para la ejecución interactiva, ya que incluso una sola sentencia puede implicar una gran cantidad de cálculos. Sin embargo, hay dos razones por las cuales los lenguajes dinámicos son menos adecuados para la construcción de grandes programas que se hayan de usar repetidamente en un entorno de producción. La primera de estas razones es que la elección de la compilación resulta más atractiva en esta situación, ya que el compilador puede generar un código objeto muy eficaz y que se puede guardar para poder usarlo posteriormente. La segunda razón es que el lenguaje en sí mismo favorece la producción de programas más fiables. El programador encuentra más facilidades para seguir la lógica del programa y el traductor puede ayudar haciendo extensas comprobaciones en el programa para asegurar su corrección.

Este es un ejemplo importante de cómo una característica del lenguaje (las reglas de ámbito) puede producir efectos que van más allá de las fronteras estrictas del lenguaje de programación. No sólo puede afectar al estilo de la programación y eficacia de los programas producidos, sino que también puede requerir una implementación totalmente diferente y un conjunto de herramientas de programación igualmente diferentes.

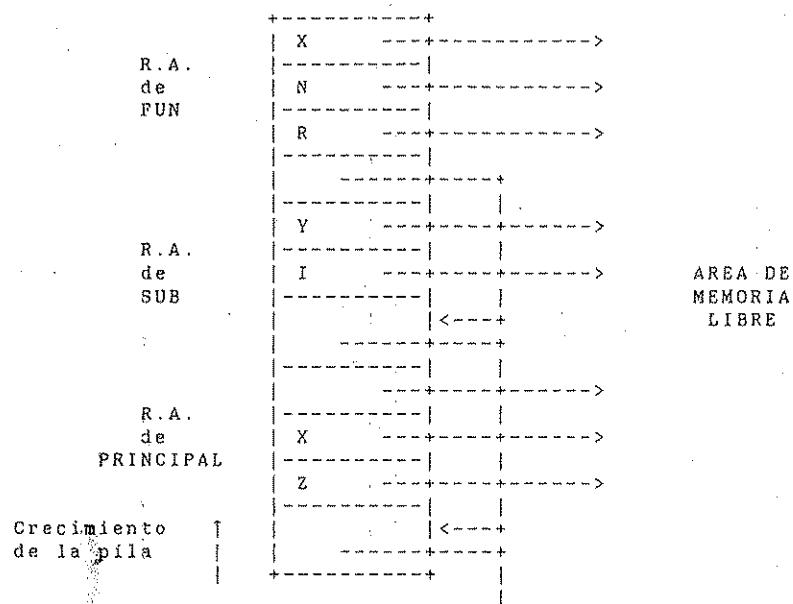


Figura 3.11 Pila de ejecución en un punto determinado del programa de la Figura 3.10.

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

Hemos estudiado de una manera informal la semántica de los lenguajes de programación mediante la descripción del comportamiento de un procesador para el lenguaje en cuestión. Intencionadamente, esta descripción operacional la hemos hecho de una manera un tanto informal. Los lectores que estén interesados en un estudio formal de la semántica operacional lo encontrarán en (Wegner 1972). Otros estudios sobre la especificación formal de la semántica se presentan en (Tennent 1976), (Gordon 1979), (Björner y Jones 1978) y (Hoare 1969). (Hoare y Wirth 1973) y (Alagić y Arbib 1978) dan una definición formal de Pascal basada en la teoría de (Hoare 1969).

Nuestra visión de la semántica está más orientada a la implementación de lenguajes. Hemos hecho especial hincapié en los importantes conceptos de ligadura y tiempo de ligadura. Este punto de vista también es compartido por otros libros de texto sobre lenguajes de programación, tales como (Elson 1973) y (Pratt 1975).

Los resultados de los experimentos expuestos en (Gannon y Horning 1975) y (Gannon 1977) mantienen la afirmación de que

aquellos lenguajes que ligan estéticamente un tipo a las variables producen programas más fiables.

Hemos resaltado también la representación de programas en tiempo de ejecución. Para encontrar más detalles adicionales sobre esta materia, consultense libros de texto sobre compiladores tales como (Gries 1971), (Bauer and Eikel 1976), (Aho y Ullman 1977) y (Barrett y Couch 1979).

En el libro (Organick y otros 1978) encontramos un estudio diferente de las descripciones de la semántica operacional de los lenguajes de programación.

Ejercicios

- 3.1 Diseñar una unidad de programa que imprima un número generado al azar cada vez que se active. La unidad no debe acceder a ninguna variable global y no recibirá parámetros. ¿Se puede resolver este problema con una unidad insensible a la historia?
- 3.2 Diseñar y ejecutar un experimento para comprobar si su compilador FORTRAN permite la implementación de subprogramas sensibles a la historia.
- 3.3 Consideremos el siguiente programa tipo ALGOL:

```

program A;
procedure B;
procedure C;

```

```
call B;
```

```
. . .
```

```
end C;
```

```
. . .
```

```
call C;
```

```
end B;
```

```
procedure D;
```

```
. . .
```

```
call B;
```

```
end D;
```

```
. . .
```

```
call D;
```

```
end A;
```

Describir cada etapa en la vida de la pila hasta que el procedimiento C llama a B. Especificar los enlaces estáticos

y dinámicos antes y después de cada llamada.

3.4 Explicar por qué los enlaces estáticos y dinámicos tienen el mismo valor en los bloques de tipo ALGOL.

3.5 Un lenguaje con reglas de ámbito dinámico puede implementar sus referencias no locales mediante una tabla que siga la pista de las variables no locales referenciables (Sección 3.7). Detalle las acciones que se han de ejecutar a la entrada y a la salida del subprograma con objeto de mantener actualizada esta tabla.

3.6 "Las ligaduras que se establecen al principio facilitan una implementación eficaz y favorecen una programación fiable; las ligaduras establecidas posteriormente proporcionan flexibilidad". Dar ejemplos que ratifiquen esta afirmación. Dar también contraejemplos.

3.7 Explicar por qué APL requiere más soporte en tiempo de ejecución que los lenguajes tipo ALGOL, y por qué estos últimos requieren a su vez más soporte en tiempo de ejecución que el FORTRAN. ¿Es este fenómeno una función del compilador/intérprete, o es inherente al lenguaje? Obtener conclusiones sobre la eficacia comparativa del tiempo de ejecución de estos lenguajes. ¿Qué ocurre con el aprovechamiento de la memoria? ¿Y con la eficacia de la programación?

3.8 En ALGOL 60, una variable puede declararse como own ("propia"). Una variable propia se asigna a la memoria la primera vez que se activa la unidad que la contiene, y su asignación de memoria permanece hasta que termina el programa. Se aplican las reglas normales de ámbito de modo que la variable sólo es conocida en la unidad donde haya sido declarada. Resumiendo, el efecto de la declaración own es ampliar la vida de la variable para que cubra toda la ejecución del programa.

a. Las variables propias no se inicializan por defecto. Esto, en la práctica, limita en gran medida su utilidad. Para poder apreciar esta limitación, escribir una función que siga la pista del número de veces que ha sido llamada. Este valor se guarda en una variable propia y se devuelve cada vez que se llama a la función.

b. Esbozar un modelo de implementación para variables de tipo own. Para simplificar, asumir que las variables own solamente pueden ser de tipo simple (no estructuras).

3.9 Diseñar un intérprete para un minilenguaje (un lenguaje que conste de cinco tipos de sentencias o menos).

3.10 Diseñar un traductor para un minilenguaje.

**TIPOS
DE
DATOS**

Los programas de ordenador se pueden contemplar como funciones que se aplican a ciertos datos de entrada para producir unos resultados determinados. En los lenguajes convencionales de programación, esta función se evalúa a través de una secuencia de pasos que produce unos datos intermedios que se almacenan en las variables del programa. Existen diferencias fundamentales entre los lenguajes en los tipos de datos que utilizan, las operaciones aplicables a los datos, y la forma en que se pueden utilizar y estructurar los datos.

Los lenguajes de programación por lo general suministran un conjunto predefinido de tipos elementales de datos, así como mecanismos para estructurar tipos de datos más complejos a partir de los elementales. Estas características se discuten en general en todo este capítulo. Nos centraremos en los lenguajes que establecen una ligadura estática entre una variable y su tipo. En concreto, revisaremos y discutiremos las características suministradas por ALGOL 68, Pascal, SIMULA 67, CLU, y Ada.

El capítulo está organizado de la siguiente manera. La Sección 4.1 trata de los tipos predefinidos. La Sección 4.2 clasifica los mecanismos básicos suministrados por los lenguajes de programación para agrupar datos elementales en estructuras de datos complejas. La Sección 4.3 nos habla de cómo se pueden utilizar tales mecanismos para definir nuevos tipos de datos en ALGOL 68 y Pascal. La Sección 4.4 trata de las conversiones de tipo. La Sección 4.5 expone algunos aspectos espinosos en cuanto a los tipos en ALGOL 68 y Pascal, y muestra las soluciones adoptadas por el lenguaje Ada. La Sección 4.6 presenta características de los lenguajes para definir tipos abstractos de datos, poniendo énfasis en SIMULA 67 y CLU. Por último, la Sección 4.7 discute modelos de soporte.

4.1 TIPOS PREDEFINIDOS

Los lenguajes de programación soportan un conjunto de tipos predefinidos que, como se mencionó en las Secciones 2.1 y 2.2, en la mayoría de los casos reflejan el funcionamiento del hardware subyacente. Al nivel del lenguaje de programación, el concepto de tipo predefinido aparece como una forma de identificar el comportamiento abstracto de un conjunto de objetos dotado de un conjunto general de operaciones. Por lo tanto, los números enteros se pueden contemplar como un conjunto de valores digamos ... -2, -1, 0, 1, 2, ... que se pueden manipular con los operadores de sobra conocidos +, -, * y /. Un traductor para un lenguaje de programación proyecta esta visión abstracta en una implementación concreta. Por ejemplo, la implementación proyecta el objeto abstracto "25" en una determinada cadena de bits, por ejemplo, 00011001 en representación de complemento a dos. De un modo muy parecido, la suma de dos enteros se proyecta en la operación de suma en coma fija de la máquina.

Basándonos en algunos de los puntos que ya se han puesto de relieve en las Secciones 2.2 y 2.4, podemos decir que los tipos predefinidos introducen cuatro propiedades provechosas. La primera de estas propiedades concierne tanto a los lenguajes con tipos dinámicos como a los lenguajes de tipos estáticos, mientras que las otras tres solamente son características de los lenguajes de tipos estáticos.

1. Transparencia de la representación interna

El programador no tiene acceso a la cadena de bits interna que representa un valor de un tipo determinado. Tal cadena de bits se modifica como resultado de la aplicación de operaciones, pero el programador no ve la modificación como una nueva cadena de bits, sino como un nuevo valor de un tipo predefinido. La transparencia de la representación interna tiene los dos siguientes efectos positivos.

Estilo de programación

La abstracción suministrada por el lenguaje incrementa la legibilidad del programa protegiendo la representación de los objetos de las manipulaciones indisciplinadas. Esto contrasta con el hardware convencional subyacente, que no se preocupa de la protección. Cualquier objeto se contempla como una simple cadena de bits que se puede manipular por el repertorio de instrucciones de la máquina. Por ejemplo, una posición que contiene una instrucción, se puede sumar a una posición que contiene una cadena de caracteres, o incluso a otra instrucción.

Modificabilidad

Se puede cambiar la representación de las abstracciones sin afectar a los programas que las utilizan. Como consecuencia, también se aumenta la portabilidad de los programas, esto es, se pueden llevar los programas a máquinas que utilicen distintas representaciones internas para los datos.

Los lenguajes de programación soportan instrucciones para leer y escribir valores de tipos predefinidos. La mayoría de los lenguajes además suministran facilidades para formatear la salida. Las máquinas efectúan la entrada/salida interactuando con los dispositivos periféricos de una forma complicada y dependiente de la máquina (que no discutiremos en este texto). Los lenguajes de alto nivel ocultan estas complicaciones y los recursos físicos involucrados en la entrada/salida de la máquina (registros, canales, etc.).

2. La utilización correcta de las variables se puede comprobar en tiempo de traducción.

Si se precisa declaración de variables, el traductor puede detectar operaciones ilegales sobre una variable, es decir, se puede conseguir la protección de variables en tiempo de traducción.

Sin embargo, ya hemos mencionado que la comprobación de tipo estático no agota todas las comprobaciones que se pueden hacer sobre un programa. Por ejemplo, la expresión i/j se puede dar estáticamente como correcta (supongamos que i y j son ambos reales) y todavía requerirá una comprobación en tiempo de ejecución por si se intenta una división por cero. Se ofrece otro ejemplo a continuación en el punto 4.

3. Se puede hacer desaparecer la ambigüedad de los operadores en tiempo de traducción.

Hay lenguajes en los que unos pocos símbolos de operadores se pueden utilizar para representar un gran número de operadores. Por ejemplo, el símbolo "+" puede representar tanto a la suma de enteros como a la suma de reales. En un lenguaje con tipos estáticos (Sección 3.3.1), el traductor puede elegir la operación de máquina que se invocará para ejecutar $A + B$, puesto que se conocen los tipos de los operandos. Esto hace a la implementación más eficiente que en los lenguajes con tipos dinámicos, tales como APL, en el que es necesario llevar la cuenta de los tipos en los descriptores en tiempo de ejecución. A un operador cuyo significado depende del tipo de sus operandos se le denomina sobrecargado, o genérico. El operador "+" está sobrecargado porque está definido tanto para los números reales como para los enteros (y está implementado por diferentes instrucciones de máquina en la representación interna). Una utilización juiciosa de la sobrecarga puede contribuir a simplificar y facilitar el uso de un lenguaje. Por ejemplo, el disponer de dos símbolos distintos para la suma entera y real, podría hacer la programación más complicada. Sin embargo, la utilización excesiva de la sobrecarga puede conducir a programas difíciles de entender, porque un único nombre denota entidades completamente diferentes.

4. Control de precisión

En algunos casos, el programador puede asociar explícitamente a un tipo, una especificación de la precisión relativa de la representación. Por ejemplo, FORTRAN permite al usuario elegir entre números en coma fija con precisión sencilla y con doble precisión.

ALGOL 68 permite especificar la precisión (relativa) requerida mediante declaraciones tales como long real, long long real (para valores de coma flotante), y long int, long long int, (para valores de coma fija). El número de prefijos long (o short) soportados está determinado por la implementación y disponible a través de las constantes int lengths, int shorts, real lengths y real shorts. En Ada se puede controlar, además la precisión de los tipos de datos numéricos.

La especificación de la precisión se puede contemplar tanto como una optimización de espacio dirigida al traductor, como un requerimiento del mismo para insertar comprobaciones en tiempo de ejecución al monitor de valores de variables. La última característica proporciona al programador una ayuda efectiva en la estimación de la corrección de los programas. Por último, los programas con especificaciones de precisión son más fáciles de adaptar a distintas instalaciones con diferentes longitudes de palabra de memoria, ya que los cambios del programa fuente se circunscribirán a las declaraciones.

4.2 AGREGADOS DE DATOS

Los lenguajes de programación le permiten al programador especificar agrupaciones de objetos de datos elementales, e incluso, agrupaciones de agregados. Un buen ejemplo es el constructor de matrices (array), que construye agregados de elementos de tipo homogéneo. Un objeto agregado tiene un nombre único. La manipulación se puede realizar sobre un único componente elemental en cada operación, accediendo al mismo mediante una operación de selección adecuada. En algunos lenguajes es posible además asignar y comparar agregados como tal conjunto.

Las Secciones 4.2.1 a 4.2.4 clasifican los mecanismos básicos de estructuración suministrados por los lenguajes de programación.

4.2.1 Producto Cartesiano

El producto cartesiano de n conjuntos A_1, A_2, \dots, A_n , y que denotaremos por $A_1 \times A_2 \times \dots \times A_n$, es un conjunto cuyos elementos son n -tuplas (a_1, a_2, \dots, a_n) , donde $a_i \in A_i$. Por ejemplo, los polígonos regulares se pueden caracterizar por un número entero, el número de lados, y un número real, la longitud de un lado. Todo polígono regular así expresado sería un elemento del producto cartesiano enteros \times reales.

A los productos cartesianos se les denomina registros en

COBOL y en Pascal, y estructuras en PL/I y en ALGOL 68.

Los lenguajes de programación contemplan los objetos del producto cartesiano como compuestos de un número de campos calificados simbólicamente. En el ejemplo anterior, las variables de tipo polígono se podrían declarar como compuestas por un campo entero (numero_de_lados) que contiene el número de lados, y un campo real (tamaño_del_lado) que contiene la longitud de cada lado. Los campos de los productos cartesianos se seleccionan especificando, en una notación sintáctica adecuada, los selectores (o nombres de campos) numero_de_lados y tamaño_del_lado. Por ejemplo, siguiendo la sintaxis de Pascal, para hacer que el polígono t1 sea un triángulo equilátero de lado 7.53, escribiríamos

```
t1.numero_de_lados := 3;
t1.tamaño_del_lado := 7.53;
```

en ALGOL 68, esto mismo se podría hacer en una sola sentencia,

```
t1 := (3,7.53)
```

4.2.2 Aplicación Finita

Una aplicación finita es una función de un conjunto finito de valores de un tipo de dominio TD sobre valores de un tipo de imagen TI. Los lenguajes de programación poseen un constructor array que nos permite definir aplicaciones finitas. La declaración Pascal

```
var a: array[1..5] of real
```

se puede contemplar como una aplicación de los números enteros en el intervalo 1 a 5, sobre los números reales.

Se puede seleccionar un objeto del conjunto imágenes con la utilización de un índice, esto es, suministrando como índice un valor adecuado dentro del dominio. Por lo tanto, la notación Pascal a[k] se puede ver como una particularización de la aplicación anterior para el argumento k. Cuando se utiliza un índice con un valor fuera del dominio, se produce un error que por lo general sólo se puede detectar en tiempo de ejecución.

En algunos lenguajes tales como APL, ALGOL 68 y Ada, los índices se pueden utilizar para seleccionar más de un elemento del conjunto imágenes. Por ejemplo, en ALGOL 68, a[3:5,5:12] especifica una submatriz bidimensional de a que contiene 24 elementos.

La estrategia para ligar el dominio de la función a un subconjunto específico de valores del tipo TD varía según el lenguaje. Basicamente, hay tres elecciones posibles.

1. Ligadura en tiempo de compilación

El subconjunto se determina cuando el programa está escrito, en tiempo de compilación. Esta restricción fué adoptada por FORTRAN, C y Pascal. Soporta variables estáticas y semiestáticas.

2. Ligadura en el momento de la creación del objeto.

El subconjunto se fija en tiempo de ejecución, cuando se crea una encarnación de la variable. Esta elección (matrices dinámicas) fué utilizada inicialmente por ALGOL 60 y ha sido adoptada por SIMULA 67 y Ada. Siguiendo la terminología introducida en el Capítulo 3, tales variables se denominan semidinámicas.

3. Ligadura en el momento de la manipulación del objeto.

Esta es la más flexible pero también la elección más costosa en términos de tiempo de ejecución. En cualquier momento del tiempo de vida del objeto se puede cambiar el tamaño del subconjunto (matrices flexibles). Esto es típico de los lenguajes interpretativos tales como SNOBOL4 y APL. De los lenguajes basados en compilador, ALGOL 68 fue el primero que lo permitió, (en la forma de matrices flex); posteriormente también lo hizo CLU. De acuerdo con la terminología introducida en el Capítulo 3, tales variables se denominan dinámicas.

4.2.3 Secuencia

Una secuencia consiste en un número cualquiera de datos elementales (componentes) de un cierto tipo TC. Este mecanismo de estructuración tiene como propiedad importante que deja sin especificar el número de componentes, y por lo tanto requiere que la representación interna sea capaz de almacenar objetos de tamaño arbitrario, (al menos en principio).

Las cadenas de caracteres son un ejemplo claro de secuencia en que el tipo del componente es el carácter. El concepto de secuencia, además recoge la idea de fichero secuencial, tan familiar dentro del proceso de datos.

Es difícil abstraer un comportamiento general de los ejemplos de secuencias suministradas por los lenguajes de programación existentes. Por ejemplo, SNOBOL4 contempla las cadenas de caracteres como objetos con un conjunto rico de operaciones. Por el contrario, PASCAL y C contemplan las cadenas simplemente como vectores de caracteres, sin primitivas especiales para manipular la cadena. Aceptando un compromiso intermedio, PL/I y Ada suministran primitivas de manipulación de cadenas de caracteres pero, para reducir el problema de la asignación dinámica de memoria, requieren que se especifique el

tamaño máximo de la cadena en la declaración de la misma. Los ficheros presentan problemas más serios, en los que a menudo se observan aspectos peculiares dependientes del sistema, como resultado de la conexión (o dependencia) con el sistema operativo.

Los operadores convencionales sobre las cadenas de caracteres incluyen los siguientes:

1. Concatenación. La concatenación de ESTE_ES_ y UN_EJEMPLO da ESTE_ES_UN_EJEMPLO.
2. Selección del primer (último) componente. La selección del último componente de la cadena anterior da 0.
3. Troceado. Se puede extraer una subcadena de otra dada especificando la posición del primer y último caracteres deseados.

Para los ficheros se suelen suministrar primitivas sencillas. Por ejemplo, un fichero Pascal solamente se puede modificar añadiendo un nuevo valor al final del fichero existente, y la lectura es sólo posible a través de una exploración secuencial.

4.2.4 Recursión

Un tipo de dato recursivo T puede contener componentes que pertenezcan al mismo tipo T. Para definir un tipo recursivo, se puede utilizar un nombre de tipo en vez de la definición del tipo. Por ejemplo, el tipo árbol binario se puede definir como vacío o como un conjunto de tres elementos, un elemento atómico, un árbol binario (izquierdo), y un árbol binario (derecho).

La recursión es un mecanismo de estructuración que se puede utilizar para definir agregados cuyo tamaño puede crecer arbitrariamente y cuya estructura puede tener una complejidad arbitraria. En contraposición a la secuencia, permite al programador crear caminos de acceso arbitrarios para la selección de componentes.

Los objetos de tipo recursivo se implementan utilizando punteros. Cada componente especificado que pertenece al tipo recursivo se representa por una posición que contiene un puntero al objeto, en vez de por el objeto mismo. Es preciso hacerlo así, ya que los objetos pueden ser de tamaño arbitrario. Por ejemplo, cada nodo de un árbol binario tiene dos posiciones asociadas: una que contiene un puntero al subárbol izquierdo, si lo hay; y otra que contiene un puntero al subárbol derecho (si lo hay). Ignoramos la información que se almacena en cada nodo. El árbol como tal, se identifica por otra posición que contiene un puntero al nodo raíz del árbol. Comenzando desde esta posición, es posible acceder a cada nodo siguiendo una cadena adecuada de punteros. Un puntero nulo se corresponde con un (sub)árbol vacío.

Las Secciones 4.3.1.2.3 y 4.3.2.2.4 muestran cómo se pueden definir árboles binarios en ALGOL 68 y Pascal, respectivamente.

4.2.5 Unión Discriminada

La unión discriminada es un mecanismo de estructuración que especifica que se va a realizar una elección entre diferentes estructuras alternativas. Cada estructura alternativa se denomina variante.

COBOL soporta uniones discriminadas con su cláusula REDEFINES. Este constructor soporta situaciones comunes en proceso de datos, en que las estructuras de algunos registros almacenados son idénticas para la mayor parte de los mismos, diferiendo únicamente en algunos campos. En un programa de nóminas, un campo puede referirse al salario mensual del empleado o a su salario por hora, dependiendo de la forma en que se le pague. En el siguiente ejemplo, SALARIO y TANTO-POR-HORA se refieren al mismo campo del registro, pero cada uno tiene una descripción diferente (PICTURE, que en COBOL es análoga al concepto de tipo).

01 REGISTRO-EMPLEADO.

05 NOMBRE	PIC X(20).
05 SALARIO	PIC 9999.
05 TANTO-POR-HORA	REDEFINES SALARIO

La mayoría de los lenguajes modernos le permiten al programador definir, en mayor o menor grado, el tipo de una variable como una unión discriminada. Dos ejemplos son la unión de ALGOL 68 y el registro con variante de Pascal, que se discutirá más adelante.

4.2.6 Conjunto Potencia

Frecuentemente es útil definir variables cuyo valor puede ser cualquier subconjunto de un conjunto de elementos de un determinado tipo T. El tipo de tales variables es el conjunto potencia (T), es decir, el conjunto de todos los subconjuntos de los elementos de tipo T. Al tipo T se le denomina tipo base. Por ejemplo, supongamos un procesador de lenguaje que acepta el siguiente conjunto O de opciones:

- LSTR: produce un listado del programa fuente.
- LSTO: produce un listado del programa objeto.
- OPTM: optimiza el código objeto.
- SFTF: salva el programa fuente en memoria masiva.
- SOBJ: salva el programa objeto en memoria masiva.
- EJEC: ejecuta el código objeto.

Un comando para el procesador puede ser cualquier subconjunto de O, tales como:

```
(LSTR,LSTO)
(LSTR,EJEC)
(OPTM,SOBJ,EJEC)
```

El tipo de comando es conjunto potencia (O).

Las variables del tipo conjunto potencia (T) representan conjuntos. Las operaciones permitidas sobre tales variables son las operaciones normales sobre los conjuntos, tales como la unión y la intersección. También es posible comprobar si un objeto dado del tipo T está en el conjunto. La utilización por parte de Pascal de este concepto, se muestra en la Sección 4.3.2.2.3.

4.3 TIPOS DEFINIDOS POR EL USUARIO

Los constructores revisados en las secciones previas permiten al programador definir objetos complejos como agregados de entidades elementales. Un ejemplo de una declaración de una variable estructurada en Pascal es:

```
var a: record x: integer;
      y: array[1..10] of char
      end
```

El tipo de la variable a no tiene un nombre explícito, sino que está descrito en términos de su representación (un producto cartesiano, uno de cuyos campos es una aplicación finita).

Algunos lenguajes modernos, tales como ALGOL 68, Pascal y Ada, además suministran una facilidad para definir un nombre de tipo nuevo. El programador puede definir tipos de datos renombrando los tipos existentes o uniendo varios tipos elementales y/o tipos definidos por el usuario, por medio de algunos de los constructores discutidos en la Sección 4.2. La noción de tipo se utiliza en estos lenguajes en un marco limitado, lo justo para obtener un mecanismo uniforme para acceder a los componentes de los objetos estructurados. Una declaración de tipo define un prototipo de estructura de datos, un arquetípico, que se puede utilizar para declarar tantas variables de este tipo como se necesiten.

Por ejemplo, en Pascal se puede declarar el siguiente tipo de producto cartesiano:

```
type complejo = record radio : real;
              angulo: real
              end
```

Todas las variables de tipo complejo están compuestas por dos campos, radio y angulo, para soportar el valor absoluto y el argumento de un número complejo. La declaración:

```
var c1,c2,c3 : complejo
define tres variables complejas denominadas c1, c2 y c3.
```

Las ventajas básicas de proveer facilidades para dar nombres explícitos a los tipos son:

1. Legibilidad

La elección apropiada de los nuevos nombres de tipo puede mejorar la legibilidad de los programas. El proceso de refinamiento paso a paso que lleva a la definición de una clase de datos, se refleja por la estructura jerárquica de las definiciones de tipos.

Por ejemplo, después de la declaración anterior de complejo, se pueden declarar los siguientes tipos:

```
voltaje = complejo;
tabla_voltaje = array[1..10] of voltaje
```

Estas declaraciones muestran de una forma clara que las variables de tipo tabla_voltaje pueden representar los valores del voltaje en un espacio de 10 puntos, estando representado voltaje por un número complejo.

2. Modificabilidad

Un cambio en las estructuras de datos que representan las variables de un tipo dado, requiere cambios únicamente en la declaración de tipo, no en las declaraciones de todas las variables (es decir, está localizado en partes pequeñas del programa). Sin embargo, a veces puede requerirse un cambio en las instrucciones del programa que manipulan las variables cuyo tipo se ha cambiado.

3. Factorización

La definición de un prototipo de una estructura de datos complicada se escribe una sola vez y después, se puede utilizar tantas veces como sea necesario para declarar variables. Esto reduce la cantidad de código necesario para copiar la misma definición para cada variable y reduce la posibilidad de errores de copia.

4. Comprobación consistente

La posibilidad de definir tipos nuevos, le permite al programador extender la aplicación de una herramienta de validación simple pero efectiva, como es la comprobación de tipo, desde la clase limitada de tipos predefinidos a otra clase de tipos definidos por el usuario. El grado de comprobación de tipos que se puede hacer depende de la noción de compatibilidad o equivalencia de tipos especificada por el lenguaje. El mecanismo de comprobación de tipos trata dos tipos compatibles como si fuesen del mismo tipo. Este tema se discutirá en la Sección 4.5.2.

En las Secciones 4.3.1 y 4.3.2 se revisarán los tipos definidos por el usuario que suministran ALGOL 68 y Pascal. En la Sección 4.6 se discuten mecanismos más potentes que permiten al programador definir tipos abstractos de datos.

4.3.1 Estructura de Tipos de ALGOL 68

La estructura de tipos de ALGOL 68 es rica y elaborada. Uno de los objetivos de diseño del lenguaje, llevado a cabo y claramente visible en su estructura de tipos, es la ortogonalidad. Para lograr simplicidad, el lenguaje suministra un número pequeño de conceptos de primitivas independientes; para lograr potencia de expresión, estos conceptos se pueden aplicar ortogonalmente, es decir, independientemente o en cualquier combinación.

ALGOL 68 utiliza una terminología precisa aunque poco usual. Con el objetivo de uniformidad, describiremos el lenguaje utilizando términos y conceptos desarrollados en el Capítulo 3.

El lector debe ser consciente de que algunos de nuestros términos (p.ej., "nombre") pueden tener un significado distinto en la terminología oficial ALGOL 68.

ALGOL 68 utiliza el término "modo" para "tipo". Una declaración de variable tiene la forma:

```
type nombre_variable
```

donde type puede ser un tipo definido por el lenguaje o definido por el usuario. Hay cinco tipos primitivos (modos sencillos) y cinco formas de construir nuevos tipos. Los tipos definidos por el lenguaje también incluyen algunos tipos no primitivos que han sido definidos por el lenguaje en términos de tipos primitivos y las facilidades de construcción de tipos.

4.3.1.1 Tipos Primitivos (Modos Sencillos)

Hay cinco tipos primitivos, de los cuales cuatro son muy corrientes: int, real, bool y char. Los números enteros se representan por int, y los reales por real. Hay dos constantes dependientes de la implementación, max int y max real, que representan respectivamente el entero y el real más grande representables. Las variables lógicas se representan por bool, que consiste en dos valores, true y false. Los caracteres elementales se representan por char.

El quinto modo es void, cuyo único valor es empty. El propósito del modo void es dotar de consistencia al lenguaje. Cada sentencia tiene un modo, incluyendo las sentencias de asignación y las llamadas a procedimientos. El modo de una sentencia de asignación es el del valor de la parte izquierda; el modo de una sentencia de llamada es el del valor devuelto; si el procedimiento no devuelve un valor, la llamada tiene el modo void.

4.3.1.2 Tipos no Primitivos (Modos Complejos)

El lenguaje suministra cinco maneras de construir nuevos modos a partir de los modos sencillos. Al nuevo modo se le puede dar un nombre o simplemente utilizarse en las declaraciones de variables sin nombre.

Por ejemplo, si tomamos X para la definición de un modo nuevo, podríamos darle el nombre nuevomodo con la siguiente definición:

```
mode nuevomodo = X
```

y utilizar posteriormente nuevomodo para declarar variables:

```
nuevomodo x,y,z
```

O podríamos poner

```
X x,y,z
```

sin dar un nombre al modo. Observar que cuando se define un modo nuevo, pasa a formar parte del lenguaje (para este programa), y por lo tanto su nombre aparece en negrita.

Los ejemplos expuestos son mecanismos de construcción de tipos de ALGOL 68.

4.3.1.2.1 Referencias

Una variable tiene dos atributos que son su valor y la referencia al área de memoria donde se almacena su valor (Sección 3.3). La mayoría de los lenguajes no ponen énfasis en la distinción entre estos dos conceptos.

Por ejemplo, en la sentencia Pascal

```
x := x+2
```

asumiendo que x se ha declarado como integer, en la parte derecha x representa el valor de la variable, mientras que en la parte izquierda x significa la referencia a la posición de memoria donde se almacena el valor. Y en la declaración

```
var x: integer
```

integer califica los valores que se pueden asignar a la variable x. No es correcto decir "el tipo de x es entero"; es más correcto decir "el tipo de los valores asignados a x es entero". Sin embargo, en la mayoría de los lenguajes se pueden utilizar indistintamente las dos frases entrecomilladas.

ALGOL 68 hace esta distinción entre valor y referencia a un objeto de una forma explícita. Una variable siempre hace alusión a la referencia al objeto. El efecto de la siguiente declaración ALGOL

```
int x
```

es declarar x como una referencia a objetos del tipo int, es decir, el tipo de x es ref int.

Ahora, para una sentencia

```
x := x+2
```

ALGOL 68 pone de manifiesto que la x de la parte derecha es simplemente un mecanismo (dereferencing) destinado a suministrar el valor necesitado por la operación "+", mientras que en la parte izquierda se trata de una referencia al lugar ocupado por un objeto y no tiene sentido acceder a su valor. El que se necesite un tipo u otro de referencia se deduce del contexto, como en el ejemplo anterior.

Observar que el efecto de la declaración o de la sentencia de asignación es el mismo que en Pascal. La diferencia está simplemente en la distinción clara que ALGOL 68 hace entre referencias y valores.

El concepto de referencia se utiliza como un mecanismo para definir nuevos modos. Obtenemos un nuevo modo sin más que precediendo con el símbolo ref a un modo definido previamente. Por ejemplo, la declaración

```
ref int ri
```

crea ri del modo ref ref int, esto es, el valor almacenado en el área referenciada por ri es del modo ref int, una referencia a un entero. En otras palabras, ri es lo que en otros lenguajes se denomina un puntero.

Podemos asignar a ri nombres del modo ref int, tal como x:

```
ri := x
```

Aquí, ri requiere un valor del modo ref int, el mismo que x. Por lo tanto, no es necesario extraer el valor de x, ri simplemente almacenará la referencia al objeto ligado a x.

La regla general para una sentencia de asignación es que el modo de la parte izquierda se utiliza para determinar si es necesaria una "extracción del valor asociado", o cualquier otra conversión en la parte derecha. De hecho, la variable de la parte izquierda va a ser una referencia a algún modo x. Si la parte derecha no es del modo al que se refiere la parte izquierda (p.ej., x), se aplica automáticamente la "extracción del valor" hasta que se efectúa la asignación.

Utilizando un puntero tal como ri , podemos acceder tanto al valor ref int como al valor int (el valor almacenado por el objeto referenciado por el valor de ref int).

Por ejemplo

```
int x,y;
ref int ri;
x := 2;
ri := x;
y := ri;
```

asignará 2, valor de x, a y. En la última asignación, y requiere un valor de modo int; a ri se le aplica dos veces la "extracción de valor" para obtener el valor requerido; en este caso, este valor es el asignado más recientemente a x. La asignación:

```
ri := 1
```

no es válida porque ri requiere un valor del modo ref int y 1 es del modo int. En general, la parte derecha debe ser de un modo con una ref menos que en la parte izquierda, o poder ser obtenido por "extracción de valor". En nuestro caso, 1 no lo es.

Podemos asignar un valor al objeto apuntado actualmente por ri especificando una "extracción de valor" sobre ri . La conversión explícita se denomina proyección (casting) en ALGOL 68. La sentencia:

```
(ref int)ri := 1
```

produce la recuperación de un objeto ref int desde ri y luego la asignación de 1 a tal objeto. En este caso, proyectando ri a ref int produciremos una referencia al mismo objeto referenciado por x, puesto que $ri := x$ es la asignación más reciente a ri . El valor de este objeto se cambia a 1.

La proyección se puede utilizar también en la parte derecha. Por ejemplo,

```
(ref int)ri := (int)ri+1
```

suma 1 al valor del objeto apuntado por ri . Sin embargo, en este caso no se requiere la proyección en la parte derecha, puesto que del contexto se podría haber deducido la necesidad de la "extracción de valor", y haberla aplicado automáticamente.

Puesto que cualquier modo se puede preceder por ref, todos los modos siguientes, así como otros, son válidos en ALGOL 68

```
ref int
ref ref int
ref ref ref real
```

Podemos crear tantos niveles de indirección como queramos. Este es un ejemplo de la ortogonalidad del diseño. No hay excepciones tales como "solo una ref puede preceder a un modo".

Un problema asociado con la utilización de punteros en la mayoría de los lenguajes suele ser el de la referencia suelta. Una referencia suelta es un puntero (es decir, una referencia), que apunta a un área de memoria que ha sido liberada. Tales problemas pueden suceder si permitimos punteros que se refieran a variables del programa. Las variables de un programa desaparecen a la salida de la unidad donde han sido declaradas localmente; por consiguiente, un puntero a ellas puede resultar perjudicial.

ALGOL 68 tiene una regla sencilla que previene de la aparición de referencias sueltas: en una asignación a una variable de referencia, el ámbito del valor que se va a asignar debe ser el mismo, o incluir, al ámbito de la variable de referencia.

El siguiente fragmento de un programa muestra algunas asignaciones legales e ilegales.

```
begin ref int ri:int i;
```

```
begin ref int rx; int x;
```

```
rx := x;                      # legal #
rx := i;                      # legal #
ri := i;                      # legal #
ri := x;                      # illegal #
end
```

en este punto, rx y x han desaparecido, no deberían existir punteros a ellas; sin embargo ri e i todavía existen
end

Como ya se trató en el Capítulo 3, los punteros en los lenguajes de programación se pueden utilizar para apuntar a objetos anónimos. En ALGOL 68, tales objetos se pueden crear mediante los generadores. Un generador crea un objeto de un modo dado y produce una referencia al objeto. Hay dos tipos de generadores: uno crea variables de pila (loc) y el otro, variables de memoria libre (heap).

La sentencia

```
ri := heap int
```

crea un objeto entero en la memoria libre y asigna su dirección a ri, que podría ser una ref int.

La sentencia

```
ri := loc int := 2
```

crea una variable local, en la pila, le asigna el valor 2, y hace que ri (una ref int) le apunte. Como siempre, este objeto dejará de existir a la salida del ámbito en que se declaró. La diferencia con las variables de programa radica únicamente en que no tienen nombre explícito.

4.3.1.2.2 Múltiplos

Múltiplos o múltiplo es el término utilizado en ALGOL 68 para designar una matriz, y se puede utilizar para crear aplicaciones finitas. El dominio de la aplicación finita es un subrango de los enteros. Los nuevos modos se denominan filas. Por ejemplo, "[]int" es el modo "fila de enteros" (una matriz unidimensional), "[,]int" es el modo "fila fila de enteros" (una matriz bidimensional), y así sucesivamente. También podemos tener el modo "[,] [] int", que es el modo "fila de fila de enteros". La diferencia entre los valores del modo "fila fila de enteros" y "fila de fila de enteros" es que el primero es una matriz bidimensional, mientras que el último es una matriz unidimensional, cada uno de cuyos elementos es a su vez una matriz unidimensional.

Podemos tener múltiplos de valores de cualquier modo. Por ejemplo,

```
[] ref [,] int
```

es un modo válido: fila de referencias a una fila fila de enteros, o un vector de punteros a una matriz bidimensional de enteros.

Hay que observar que aunque el número de dimensiones de una matriz es parte de su tipo, el tamaño de una matriz, es decir, el número de elementos de cada dimensión, no lo es. Sin embargo, cualquier variable declarada como múltiplo debe indicar este

tamaño, al menos en términos de variables.

```
[m:n] int ri
```

declara que el objeto referenciado por ri va a ser una matriz semidinámica con sus índices dentro del rango m..n. Cuando se encuentra esta declaración en tiempo de ejecución, se asigna memoria para la matriz de acuerdo a los valores actuales de m y n; ri conserva este tamaño hasta que desaparece a la salida de su ámbito.

En otras palabras, las matrices se pueden declarar como flexibles, en cuyo caso su tamaño se puede cambiar cuando se efectúa una asignación sobre ellas. Son un ejemplo de lo que hemos denominado variables dinámicas.

```
flex[1:0] int a
```

declara que el objeto referenciado por a va a ser una fila que inicialmente no contiene enteros.

```
a := (2,3,4)
```

cambia los límites a [1:3] y asigna valores a todos los elementos de a. Los límites sólo se pueden cambiar mediante una asignación a la totalidad de la matriz.

El modo string está predefinido por el lenguaje como un vector flexible de caracteres.

```
mode string = flex[1:0] char
```

Las acciones permitidas sobre las cadenas de caracteres, que son las mismas que para las matrices, son la indexación, con la que se puede obtener un elemento, y el recorte con el que se puede obtener una sección (o "rebanada" en la jerga de ALGOL 68) de la matriz.

4.3.1.2.3 Estructuras

Las filas permiten la creación de una aplicación finita, es decir, un modo consistente en elementos homogéneos. También se puede construir un nuevo modo de elementos heterogéneos mediante un producto cartesiano. Esto se puede hacer mediante la estructura de ALGOL 68. Una estructura, o valor estructurado, consiste en un conjunto de valores, denominados campos, que pueden tener modos diferentes.

```
mode persona = struct(string nombre, int edad)
```

crea un modo nuevo compuesto por dos campos nombre y edad. Teniendo creadas variables de este modo, podemos asignarles valores.

```
persona mama, papa;
mama := ("elena", 35);
papa := ("tomas", 36)
```

Además se puede acceder a los campos individuales para recuperar o asignar un valor, utilizando el selector of, según se ve en el ejemplo siguiente:

```
edad of papa = edad of mama + 1
```

Se pueden definir estructuras de cualquier modo:

```
struct (ref ref [], int uno, flex[1:0] real dos,
       int tres, persona cuatro)
```

Esta estructura es un modo permitido, aunque obviamente no muy útil.

Se pueden crear estructuras de datos recursivas mediante la combinación de referencias y estructuras. Por ejemplo, el modo de los nodos de un árbol binario se puede declarar como:

```
mode nodo_arbol_binario = struct(string informacion,
                                    ref nodo_arbol_binario izquierdo,
                                    derecho)
```

Los nodos predefinidos compl, bits y bytes, y sema se han definido utilizando las estructuras:

```
mode compl = struct(real re, im);
mode bits = struct([1:bits width] bool x);
mode bytes = struct([1:bytes width] char x);
mode sema = struct(ref int x)
```

Las constantes definidas en implementación bits width y bytes width indican respectivamente el número de bits y de bytes de una palabra de la máquina. El modo sema es el semáforo (ver Sección 4.5.2.4.1) y se puede utilizar para programación concurrente. Están definidas las operaciones up y down sobre una estructura sema. Los nombres de los campos de las estructuras bits, bytes y sema no son accesibles al programador, así, por ejemplo, no se puede utilizar indexación con un valor bits.

El modo bits se puede utilizar para implementar conjuntos potencia que no están directamente soportados por el lenguaje (ver Ejercicio 4.17).

4.3.1.2.4 Uniones.

En una ingeniosa desviación respecto de los lenguajes anteriores de tipo ALGOL, ALGOL 68 permite las uniones discriminadas, en la forma de modos unidos.

```
mode ent_log = union(int, bool)
```

define un modo nuevo ent_log, cuyos valores pueden ser enteros o lógicos. Observar que la variable x declarada como:

```
ent_log x
```

en todo momento contiene un valor bien de tipo int o de tipo bool, pero el tipo de x es siempre ref ent_log.

Las asignaciones a una variable x de tipo ent_log son igual que las demás sentencias de asignación:

```
ent_log x,y;
y := 5;
x := y;
x := true
```

Sin embargo, el acceso al valor de x, no se puede hacer tan fácilmente, debido a que el tipo del valor se debe establecer antes de poder utilizar el valor. La determinación del tipo se hace con la utilización de una cláusula de conformidad. La cláusula de conformidad asegura que no sucedan incompatibilidades de tipo en tiempo de ejecución, es decir, se le fuerza al programador a anticiparse a tales eventos.

```
case x in
  (int x1):....x1....
  (bool x2):....x2...
esac
```

es un ejemplo de tal cláusula. Dentro de cada alternativa de la cláusula case, se establece el tipo de x y se utiliza un nombre nuevo para referirse a x. Más concretamente, x1 y x2 son constantes "inicializadas" con el valor de x a la entrada de la cláusula de conformidad; sus tipos son entero y lógico respectivamente, y por tanto, la comprobación de tipos en el uso de x1 y x2 se puede hacer estáticamente.

Los modos unidos pueden derivarse de cualquier otro modo, incluyendo el void, tal y como:

```
union(integer, ref[]int, void)
```

Los modos unidos se utilizan frecuentemente en procedimientos que pueden recibir parámetros de distintos tipos. Representan la introducción de un concepto importante, aunque peligroso, dentro de un lenguaje de programación. De esta forma elegante, sistemática y segura, se ofrece la unión discriminada.

4.3.1.2.5 Procedimientos

El último método de construcción de tipos nuevos en ALGOL 68 se basa en la utilización de procedimientos. El concepto de modo en ALGOL 68 es más general que el de tipo en otros lenguajes. Por ejemplo, un procedimiento es un objeto de modo procedure con

modos específicos para sus parámetros y su resultado.

```
modo pl = proc(bool,real) int
```

define un modo nuevo pl, que es un procedimiento que recibe un parámetro lógico y otro real, y devuelve un resultado entero. Al igual que con otros modos, se pueden declarar y hacer asignaciones a variables del modo pl, en contraste con la mayoría de los demás lenguajes.

```
pl x,y;
x := proc(bool b, real r) int: ... cuerpo del procedimiento ...
y := x;
```

Con los procedimientos, se puede ver claramente la total generalidad de los modos de ALGOL 68 y la aplicación sistemática del diseño ortogonal. Se pueden construir modos de procedimiento a partir de cualquier otro modo, como uniones, referencias, u otros procedimientos. Por lo tanto, la siguiente definición es válida.

```
modo proc_comico = proc(int,ref[] bool)
    proc([]ref int, proc(real) int) void
```

Se define un modo que es un procedimiento que devuelve otro procedimiento como resultado; el procedimiento devuelto toma un procedimiento como segundo parámetro y devuelve un valor de tipo void.

La ortogonalidad del diseño hace que las reglas del lenguaje sean simples y uniformes. Lo que el lenguaje no hace, ni se lo ha propuesto, es impedir la codificación de construcciones complicadas. La estructura de tipos de ALGOL 68 se presenta en la Figura 4.1.

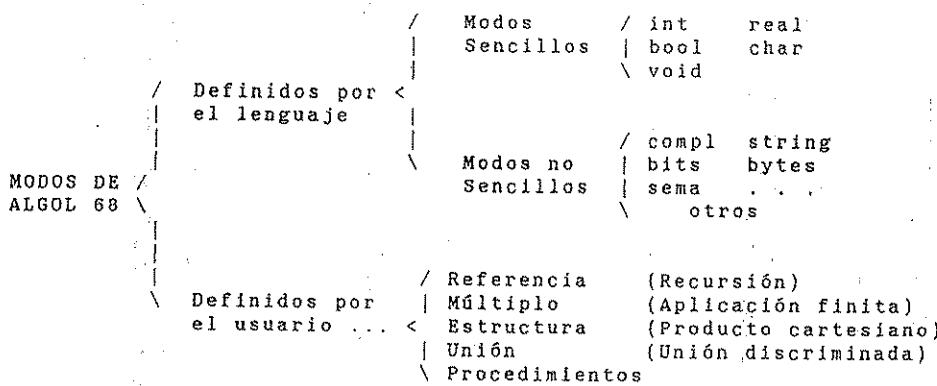


Figura 4.1 Estructura de tipos de ALGOL 68

4.3.2 Estructura de Tipos de PASCAL

4.3.2.1 Tipos no Estructurados

Como en ALGOL 68, en última instancia los tipos de datos en Pascal se deben construir a partir de componentes no estructurados pertenecientes a tipos primitivos no estructurados. Algunos de estos tipos no estructurados están predefinidos, otros son definidos por el usuario. Los tipos no estructurados predefinidos son integer, real, boolean y char. Un valor de tipo integer es un elemento de un subconjunto ordenado del conjunto de los enteros, dentro de una implementación definida. El identificador standard maxint, dentro de una implementación determinada, denota el valor entero más grande que se puede manejar por la implementación dada. Un valor de tipo real es un elemento de un subconjunto de los números racionales, dentro de una implementación determinada. Tanto la magnitud máxima como la precisión de los valores reales son dependientes de la implementación; los límites vienen impuestos por la aritmética de coma flotante de la máquina subyacente. Un valor de tipo char es un elemento de un conjunto finito y ordenado de caracteres. Tanto el conjunto de caracteres como su relación de orden están definidos en implementación. Las variables lógicas pueden tener uno de los dos valores true y false; se pueden manipular con los operadores lógicos and, or y not, y se considera que false es menor que true, por lo que se pueden comparar entre sí.

Los enteros, las variables lógicas y los caracteres, se pueden denominar colectivamente como tipos ordinales, puesto que cada elemento del tipo tiene un único predecesor y un único sucesor. Para las variables lógicas, se define que true es el sucesor de false. El sucesor y el predecesor de un valor ordinal se evalúa mediante las funciones incorporadas succ y pred. Tales funciones son ejemplos de operadores sobrecargados, puesto que pueden aceptar un parámetro de cualquier tipo ordinal.

Los programadores de Pascal pueden definir tipos ordinales nuevos de dos formas. La primera es mediante la enumeración de los valores posibles. Por ejemplo,

```
type dia = (domingo,lunes,martes,miercoles,jueves,viernes,sabado)
```

Esta declaración tiene los efectos siguientes:

1. Introduce un nuevo tipo dia;
2. Define domingo, lunes, ... sabado como las constantes de este tipo nuevo.
3. Define una relación de orden entre las constantes: domingo < lunes < ... < sabado.

Las únicas operaciones definidas implícitamente sobre las variables de este tipo son las asignaciones y las comparaciones. Por lo tanto, después de la declaración de las variables.

```
var hoy, mañana, ayer, mi_cumpleaños : dia
las siguientes sentencias representan manipulaciones correctas de las variables.
```

```
hoy := jueves;
mañana := succ(hoy);
ayer := pred(hoy);
mi_cumpleaños := mañana
```

Desgraciadamente, los valores de los tipos enumerados no se pueden leer ni escribir por un programa. Aunque podemos decir que el tipo `dia` incluye todos los nombres de los días de la semana, debemos leer el valor de un día en alguna forma codificada, digamos, entero, y entonces convertirla explícitamente a un valor de tipo `dia`.

La segunda forma de definir tipos ordinales nuevos es mediante la especificación del subrango de un tipo ordinal, el tipo ordinal asociado o tipo base. Por ejemplo, teniendo definido el tipo `dia`, podemos definir `dia_trabajo` como el subrango lunes a viernes. El siguiente fragmento ilustra esta característica.

```
type dia_trabajo = lunes..viernes;
edad = 0..120 (subrango 0 a 120 de los enteros);
var mi_edad : edad;
dia_clase : dia_trabajo;
```

Se pueden utilizar tipos nuevos sin darles un nombre explícito, como en la línea siguiente:

```
var plan_cursado: (bachiller,bup);
```

que especifica el tipo anónimo de `plan_cursado` como la enumeración de los valores `bachiller` y `bup`.

Las declaraciones anteriores se pueden continuar con las sentencias:

```
plan_cursado := bachiller;
mi_edad := 25;
if plan_cursado = bup
  then dia_clase := martes
  else dia_clase := viernes
```

Las variables de tipo subrango tienen el mismo comportamiento que las variables del tipo base, excepto para el conjunto de valores que pueden asumir, que es un subconjunto del conjunto de valores del tipo ordinal asociado. La verificación de esta propiedad requiere intrínsecamente una comprobación en tiempo de ejecución. Por ejemplo, la ejecución de `dia_clase :=`

`succ(dia_clase)` podría provocar un error en tiempo de ejecución si `dia_clase` es igual a viernes, ya que `succ(viernes)` es sábado, y sábado no pertenece al tipo `dia_trabajo`.

4.3.2.2 Constructores de Agregados

4.3.2.2.1 El constructor Array

El constructor `array` permite al programador definir aplicaciones finitas. La forma general de una estructura de matriz es

```
array[t1] of t2
```

donde `t1`, el tipo del índice o dominio, es un tipo ordinal; y `t2`, el tipo del componente o imagen, es el tipo de cada componente de la matriz.

Por ejemplo:

```
type sabor = (chocolate,menta,melocoton,fresa,vainilla,queso,
              tomate,ajo,cebolla);
sabor helado = chocolate..vainilla;
helado_pedido = array[sabor_helado] of boolean;
var mi_pedido,tu_pedido : helado_pedido;
escogido : sabor_helado;
```

se puede continuar con

```
for escogido := chocolate to vainilla do
  mi_pedido[escogido] := false;
  mi_pedido[menta] := true;
  tu_pedido := mi_pedido
  ( tanto mi_pedido como tu_pedido son menta )
```

Observar que el permitir acceder a una matriz como

```
mi_pedido[succ(elegido)]
```

requiere una comprobación en tiempo de ejecución con vistas a verificar que el índice tiene un valor dentro de los límites. De hecho, si `elegido = vainilla`, `succ(elegido)` generaría queso, es decir, un valor que no pertenece al tipo `sabor_helado`. Pascal contempla las matrices con diferentes tipos de índice como tipos distintos, tal como:

```
type a1 = array[1..50] of integer;
a2 = array[1..70] of integer
```

Puesto que los procedimientos requieren parámetros formales que tengan un tipo específico, no es posible, por ejemplo, escribir un solo procedimiento que sea capaz de ordenar las matrices `a1` y `a2`.

En Pascal se pueden definir matrices multidimensionales como matrices cuyos elementos son a su vez matrices. Por ejemplo:

```
type fila = array[-5..10] of integer;
var mi_matriz : array[3..30] of fila

No obstante, la abreviación

var mi_matriz : array[3..30,-5..10] of integer

también está permitida..
```

4.3.2.2 El Constructor Record

El constructor record (registro) se puede utilizar para definir productos Cartesianos. La forma general de una estructura record es:

```
record campo_1 : tipo_1;
  campo_2 : tipo_2;
  .
  .
  .
  campo_N : tipo_N
end
```

donde campo_i, siendo $1 \leq i \leq N$, es un identificador de campo, y tipo_i, siendo $1 \leq i \leq N$, es un tipo de campo. Se puede acceder al registro como un todo o a los campos individuales mediante la utilización del símbolo ":" como un selector (notación puntual). Por ejemplo, las declaraciones:

```
type poligono_regular = record no_de_lados : integer;
  long_lado : real
end;

var t,c,p : poligono_regular;
```

se pueden seguir con las sentencias:

```
t.no_de_lados := 3; t.long_lado := 7.53;
  (t.es un triangulo equilátero. La longitud del lado es 7.53)
c.no_de_lados := t.no_de_lados+1;
c.long_lado := 2*t.long_lado;
p := c
```

Un tipo registro puede además tener una parte variante, en cuyo caso es posible definir uniones discriminadas. Por ejemplo:

```
type departamento = (domestico, deportes, droguería, alimentación,
bebidas);
mes = 1..12;
artículo = record precio : real;
  case disponible : boolean of
    true : (cantidad : integer;
      donde : departamento);
    false: (mes Esperado : mes)
  end
```

El identificador de campo disponible es el componente discriminante (el campo indicador) de la estructura record anterior. Si el valor de disponible es true, entonces un artículo se caracteriza por la cantidad disponible y el nombre del departamento en el que se encuentra. Si es false, entonces un artículo se caracteriza por el mes en que se espera su recepción. En ambos casos el artículo tiene un precio asociado.

Pascal permite al programador acceder a todos los campos de una estructura registro, incluido el campo indicador. Si a1 y a2 se declaran y manipulan como sigue:

```
var a1,a2 : artículo;
```

```
a1.precio := 5.24;
a1.disponible := true;
a1.cantidad := 29;
a1.donde := bebidas;
a2.precio := 324.99;
a2.disponible := false;
a2.mes Esperado := 8
```

las estructuras resultantes se muestran en la Figura 4.2.

Generalmente la comprobación de tipo para los registros con variante sólo puede hacerse en tiempo de ejecución. Por ejemplo si a es de tipo artículo,

```
a.cantidad
```

es una selección de campo correcta sólo si la variante actual tiene un valor true en su campo indicador. La comprobación de tipo dinámica requiere llevar control de la variante actual en un descriptor en tiempo de ejecución para cada registro con variante. Desgraciadamente, la posibilidad de modificar el campo indicador y las variantes, independientemente el uno de las otras, y sin consistencia entre ellos, hace difícil de implementar la comprobación en tiempo de ejecución. Por ejemplo, si se pasa el campo indicador a un procedimiento como un parámetro modificable, el procedimiento podría actualizar el descriptor en cada asignación hecha sobre el campo de indicador. Sin embargo, el procedimiento no sabe si está cambiando un campo indicador de un registro con variante, a menos que además se le pase tal información como un estado adicional asociado con cada

parámetro actual. Cada acceso a un parámetro formal dentro del procedimiento, sea o no un campo indicador, podría entonces requerir la comprobación del estado y, posiblemente, actualizar el descriptor. Esta solución podría hacer la implementación de los procedimientos un tanto ineficientes.

precio	5.24	precio	324.99
disponible	true	disponible	false
cantidad	29	mes Esperado	8
donde	bebidas		

Figura 4.2 Un ejemplo de un registro con variante en Pascal.

Veremos más adelante que la implementación convencional de los registros con variante consiste en solapar todas las variantes sobre el mismo área de memoria. Por tanto, los registros con variante permiten al programador interpretar la cadena de bits almacenada en este área bajo distintos puntos de vista suministrados por los tipos de cada variante. En nuestro ejemplo, después de poner bajo una variante los campos cantidad y donde de la variable a1, es posible cambiar la variante poniendo disponible a false y entonces interpretar el valor previamente almacenado en cantidad como el valor del mes Esperado.

Esta es una utilización insegura, aunque legal, de los registros con variante. La contemplación del mismo área de memoria bajo distintos nombres y tipos puede ser muy potente a la hora de establecer un modelo de algunas situaciones prácticas. Por ejemplo, una unidad de programa que simula un dispositivo de entrada podría contemplar un variable como una secuencia de k caracteres, mientras que una unidad de programa que simula un procesador de propósito especial podría contemplar la misma variable como un entero que se va a leer.

Sin embargo, en general, el dar nombres y tipos distintos al mismo objeto es una práctica de programación peligrosa. El hecho de que un objeto se pueda modificar bajo un nombre y el efecto de la modificación sea visible bajo un nombre distinto hace que los programas sean particularmente difíciles de leer y escribir. Además, para contemplar una cierta cadena de bits bajo distintos tipos, se debería conocer cómo representa los distintos tipos el traductor. En el ejemplo anterior, se debería conocer que una secuencia de k caracteres y un entero ocupan la misma cantidad de memoria, es decir, una palabra. Por consiguiente, la corrección de los programas sería dependiente de la implementación; por ejemplo, el traslado del programa a una instalación en la que los enteros y los caracteres ocupasen una palabra entera haría que el programa resultase incorrecto.

Aún peor es el hecho de que el campo indicador de un registro con variante es opcional. El anterior tipo articulo se podría haber declarado como:

```
record precio : real;
  case boolean of
    true : (cantidad : integer;
              donde : departamento);
    false : (mes Esperado : mes)
end
```

En este caso, la construcción de registro con variante es intrínsecamente insegura. Tanto cantidad como donde o mes Esperado se pueden utilizar cuando no están presentes, y no hay forma de detectar el error, incluso en tiempo de ejecución, porque no hay un campo del registro que denote la variante aplicable actualmente. La solución de hacer que el compilador inserte un campo indicador adecuado, aunque factible, no tiene una justificación real, porque la ausencia del campo indicador es una elección deliberada por parte del programador para ahorrar memoria. Por lo tanto, la ausencia del campo indicador hace imposible la comprobación de tipos en tiempo de ejecución.

Es interesante contrastar las inseguridades de la unión discriminada de Pascal con la solidez conceptual de la equivalente de ALGOL 68, los modos unión. Esto muestra cómo la aparente simplicidad de Pascal, que es responsable de mucha de su popularidad, a veces oculta problemas insospechados.

4.3.2.2.3 El Constructor Set

El constructor set (conjunto) es una versión restringida del constructor conjunto potencia, ya que el tipo base solamente puede ser un tipo ordinal. Por consiguiente, no es posible definir conjuntos de reales, ni conjuntos de tablas, ni conjuntos de conjuntos, etc.

Las siguientes declaraciones definen un tipo enumerado y dos variables de tipo conjunto.

```
type vegetales = (judias, coles, zanahorias, apio, lechuga, cebolla,
  setas);
var mi ensalada, sobras : set of vegetales (variables que pueden
  contener cualquier conjunto de vegetales; su tipo es el
  conjunto potencia de vegetales);
```

Las siguientes sentencias representan manipulaciones válidas

```

sobras := . . . ;
mi_ensalada := [zanahorias..cebolla] {asigna un valor de conjunto
con cuatro miembros a mi_ensalada};
if not judias in sobras {comprobación de pertenencia}
  then mi_ensalada := mi_ensalada+sobras ("+" significa "unión
      de conjuntos")

```

4.3.2.2.4 Punteros

El tiempo de vida de las variables en Pascal está ligado a la estructura estática de la anidación de procedimientos, ya que se asigna automáticamente memoria para las variables sobre la pila cuando se activa el procedimiento al que son locales, y se libera cuando termina la activación. Pascal también suministra punteros a objetos anónimos ubicados en la memoria libre. Los objetos de memoria libre se asignan explícitamente mediante la sentencia de creación new (ver Sección 3.6.3).

El siguiente ejemplo muestra la utilización de los punteros Pascal en la construcción de tipos recursivos tales como árboles binarios.

```

type ref_arbol = ^ nodo_arbol_binario;
  nodo_arbol_binario = record info : char;
                           izquierdo,derecho : ref_arbol
                         end;
var mi_arbol : ref_arbol

```

mi_arbol se define como un puntero (\uparrow) a un nodo raíz de un árbol binario de caracteres.

Se puede construir un árbol binario vacío con la asignación siguiente:

```
mi_arbol := nil
```

Cualquier puntero puede contener el valor nil, independientemente del tipo de objeto al que apunte; ello quiere decir que no apunta a ningún elemento. Si p es un puntero distinto de nil, al objeto apuntado por p se le denota $p\downarrow$. A diferencia de ALGOL 68, las "extracciones del valor asignado" siempre se deben escribir expresamente.

Se puede crear un árbol binario compuesto por un nodo escribiendo:

```

new(mi_arbol);
mi_arbol^.info := simbolo;
mi_arbol^.izquierdo := nil;
mi_arbol^.derecho := nil

```

La primera instrucción asigna un registro de tipo nodo_arbol_binario y hace que mi_arbol se refiera a él. Las tres

instrucciones siguientes inicializan el campo info con el valor de la variable simbolo que contiene un carácter, y los dos campos de punteros a nil. El resultado se refleja en la Figura 4.3, asumiendo que simbolo es igual al carácter a.

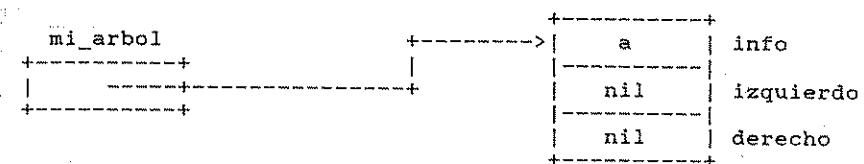


Figura 4.3 Un árbol binario con un elemento.

Podemos añadir un árbol izquierdo a mi_arbol como sigue:

```

new(ref_nodo) {ref_nodo debe ser de tipo ref_arbol};
ref_nodo^.info := 'b';
ref_nodo^.izquierdo := nil;
ref_nodo^.derecho := nil;
mi_arbol^.izquierdo := ref_nodo

```

El resultado se muestra en la Figura 4.4.

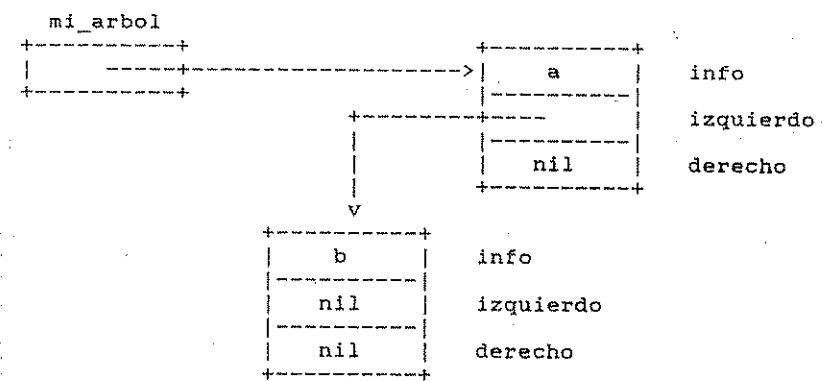


Figura 4.4 Árbol binario de la Figura 4.3 después de añadir un subárbol izquierdo.

Los punteros se pueden manipular con asignaciones y con comparaciones de igualdad y desigualdad. Sin embargo, tales manipulaciones solamente son válidas si los operandos apuntan a objetos de tipo compatible (ver Sección 4.5.2 para una definición). A diferencia de ALGOL 68, los punteros Pascal pueden apuntar sólo a objetos sin nombre; en concreto, no pueden apuntar a la posición asociada con una variable que está ubicada en la pila.

4.3.2.2.5 Datos Empaquetados

En algunas aplicaciones, por ejemplo, en programación de sistemas, al programador le gustaría poder utilizar un lenguaje de alto nivel pero manteniendo la capacidad de los lenguajes ensambladores para empaquetar toda la información que sea posible en una palabra de máquina. En Pascal, se pueden especificar estructuras de datos empaquetadas sin más que preceder array, set o record por la palabra clave packed. Obviamente, la semántica de la estructura de datos es independiente de que exista o no el prefijo packed, pero el compilador puede utilizar esta información para minimizar la cantidad de memoria asignada a la estructura de datos. Sin embargo, como veremos en la Sección 4.7, los procedimientos de acceso son más lentos para las estructuras de datos empaquetados.

Pascal trata las cadenas como vectores empaquetados de caracteres. Sin embargo, debido a que son matrices, tienen un tamaño máximo fijo durante su tiempo de vida.

4.3.2.2.6 El Constructor File

Un fichero ("file") Pascal es una secuencia de elementos de cualquier tipo. Las declaraciones siguientes definen t1 y t2 como variables de tipo fichero.

```
type patron = record ... end;
cinta = file of patron;
var t1,t2 : cinta;
```

Automáticamente se asocia con cada fichero una zona de memoria que contiene el siguiente elemento del fichero. El programa puede leer de, o escribir en, esta zona de memoria. Las operaciones get y put leen el siguiente elemento sobre la zona de memoria, y añaden el contenido de la zona de memoria al final del fichero, respectivamente. Los ficheros Pascal solamente se pueden procesar secuencialmente. La posición actual dentro de un fichero se actualiza implícitamente con las operaciones get y put.

La estructura de tipos de Pascal se puede resumir en el diagrama de la Figura 4.5, en la que se muestra claramente la jerarquía impuesta por el lenguaje.

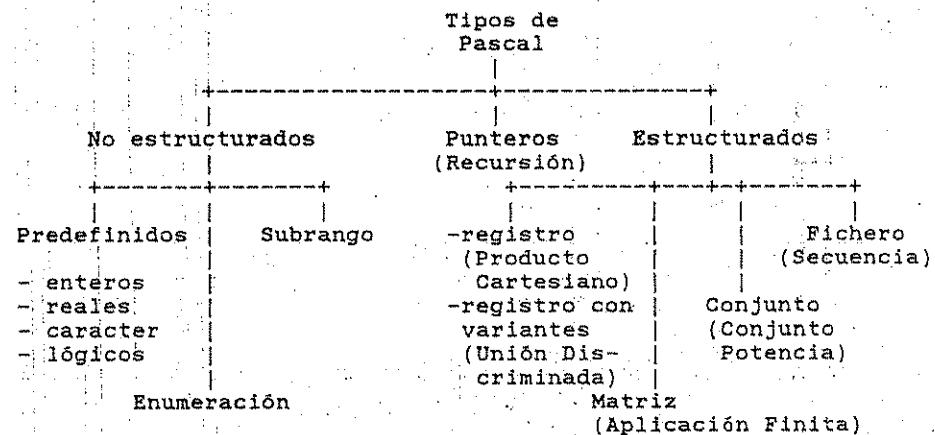


Figura 4.5 Estructura de tipos de Pascal.

4.4 CONVERSIÓN DE TIPO

Con frecuencia es necesario convertir un valor de un tipo a un valor de otro; por ejemplo, cuando queremos sumar la variable entera v con la constante real 3.753. En la mayoría de los lenguajes, tal conversión es implícita. Por ejemplo, la evaluación de la suma anterior generalmente implica una conversión, de entero a real, de la variable v y después la realización de la suma en coma flotante. La conversión implícita se hace explícitamente por el traductor, que genera código de conversión basado en el tipo de los operandos y la jerarquía de tipos del lenguaje. Por ejemplo, la jerarquía de FORTRAN es COMPLEX > DOUBLE PRECISION > REAL > INTEGER. Dada la operación a op b, el operando con un tipo menor en la jerarquía, se convierte al tipo del otro operando antes de la ejecución de op.

ALGOL 68 aplica consistentemente el principio de conversión implícita (coerciones en la jerga ALGOL 68) hasta el extremo. El tipo del valor requerido en cualquier punto de un programa ALGOL 68 se puede determinar por el contexto. Por ejemplo, si el modo de a es ref.t, el tipo del valor producido por la parte derecha de una asignación a := ... debe ser t. Está definido un conjunto de reglas que efectúan la coerción implícita de los valores de un tipo dado al tipo deseado. Ya hemos visto un ejemplo de coerción ("dereferencing"), que consistía en que un valor era automáticamente desligado de una ref. La misma idea, pero aplicada a funciones, es el "deprocedurening"; consiste en efectuar una llamada a un procedimiento y utilizar el valor devuelto; el enfilamiento ("rowing") convierte un valor elemental a una fila de valores elementales; el vaciado ("voiding") produce empty, originando la pérdida del valor original; la unión ("uniting") cambia el modo a un modo unión específico que debe incluir al modo original; la extensión ("widening") convierte un

int a real y un real a compl. No hay coerción de real a int. Esta conversión se debe realizar explícitamente, expresando el programador su intención.

Se muestran ejemplos de estas coerciones en el siguiente segmento de programa.

```
int e; real r; [1:1]int filae, ref int refe;
union(int,real)er; proc int p;

r := e/r; # extensión de i #
er := e; # union #
er := r; # union #
e := p; # llamada a procedimiento #
e := ref e # extracción de valor (dos veces) #
p; # llamada a procedimiento seguida de evacuación #
filae := 5; # enfilamiento #
```

Es interesante considerar la interacción de las reglas de coerción con otras construcciones del lenguaje. Por ejemplo, los modos unión tales como

```
mode equivoco = union(int,ref int)
```

no están permitidos, puesto que para una variable v de este modo, a veces no se puede determinar si se necesita o no extracción de valor. Por ejemplo, considerar:

```
equivoco e;
ref int v;
e := v
```

?Se debería aplicar para v la extracción de valor o no? Una variable unión puede tomar valores de distintos modos. Pero el que se aplique o no la extracción de valor se determina por el modo del valor que se desea obtener, por ejemplo, el de la parte izquierda. En la asignación mencionada anteriormente, estas dos reglas podrían entrar en conflicto, ya que la parte izquierda puede aceptar bien un int o una ref int. Por lo tanto no se permiten tales modos unión, a los que se denomina incestuosos.

Estas nociones sobre la coerción en ALGOL 68 se pueden criticar sobre la base de que conducen a programas un tanto oscuros. Las conversiones automáticas disminuyen la capacidad del traductor para la comprobación de tipos del programa, ya que anulan el tipo declarado de una variable con las transformaciones automáticas por defecto. En general, cuanto más pueda hacer implícitamente el traductor, menos servicio de comprobación de errores puede proveer, ya que el compilador puede suponer que los errores reales son peticiones implícitas de conversión.

4.5 ÁREAS PROBLEMÁTICAS Y SOLUCIONES SENCILLAS

Hemos acentuado el hecho de que los lenguajes no son importantes en sí mismos, sino solamente como herramientas que ayudan en la producción de programas. Por tanto estamos interesados no en las características que hacen a un lenguaje "más ingenioso" o imaginativo que otro, sino en cómo tales características pueden soportar la producción de un software de buena calidad global.

Esta sección repasa las características más discutibles de los lenguajes de programación vistos anteriormente y muestra cómo el descubrimiento de imperfecciones ha estimulado a la definición de diferentes, y esperamos mejores características.

4.5.1 Consistencia de Tipos.

El principal efecto de la introducción de tipos es la posibilidad de comprobación de tipos a nivel estático. Hay una mayor probabilidad de que los programas resultantes sean correctos, y además más eficientes, ya que no es necesario tener descriptores de los tipos en tiempo de ejecución ni efectuar comprobaciones sobre ellos. Se dice que un lenguaje tiene consistencia de tipos o que está orientado a una rigurosa comprobación de tipos ("strongly typed"), si permite que toda la comprobación de tipos se haga estáticamente.

Examinando Pascal y ALGOL 68, podemos concluir que Pascal no tiene consistencia de tipos, y ALGOL 68 sí, por las siguientes razones:

1. El tipo de un parámetro procedimiento o función, esto es, un procedimiento o una función que son a su vez parámetros, no es determinable en Pascal en tiempo de traducción. Por ejemplo, el procedimiento Pascal:

```
procedure quien_sabe(i,j:integer; procedure f);
  var k : boolean;
begin  k := j < i;
       if k then f(k) else f(j)
end
```

puede contener una o dos llamadas incorrectas al parámetro actual asociado con el parámetro formal f, debido a incongruencia de tipo y/o diferencia en el número de parámetros. Un compilador no puede prever un error que surgirá en tiempo de ejecución, puesto que la presencia del error depende del parámetro actual en particular con que se haya llamado al procedimiento quien_sabe.

Por el contrario, ALGOL 68 requiere de los procedimientos que son parámetros, que especifiquen los tipos (modos) de sus parámetros y sus valores

devueltos. Por ejemplo, el procedimiento `quien_sabe` en ALGOL 68 se debe escribir como:

```
proc quien_sabe=(int i,int j,proc(bool) void f) void:
begin bool k;
  k:= j<i;
  if k then f(k)
    else f(j) /*llamada ilegal, detectada por el compilador*/
  fi
end
```

2. En Pascal, los subrangos no se pueden comprobar estáticamente. Por ejemplo, en `a := b+c`, donde las variables `b` y `c` se han declarado como pertenecientes al subrango `1..10`, no es posible establecer a priori si el valor de `b+c` pertenece al tipo subrango; esto solamente se puede hacer en tiempo de ejecución. Puesto que el tipo del índice de una matriz es parte de la definición del tipo de la matriz, el índice siempre pertenece al tipo base del tipo del índice, y por tanto, el acceso a un elemento requiere una comprobación en tiempo de ejecución para verificar que el índice está dentro de los límites. Por otra parte, ALGOL 68 considera los límites de la matriz como parte no del tipo, sino del valor. Los subíndices de las variables subindexadas se ignoran en tiempo de traducción, y se inserta la comprobación sobre los valores para evaluación en tiempo de ejecución.

3. No es posible comprobar estáticamente la utilización correcta de los registros con variante de Pascal; y rara vez se suministra comprobación en tiempo de ejecución, debido al fuerte impacto sobre la eficiencia de la ejecución. Por lo tanto, los registros con variante suministran una escapatoria que permite al programador interrumpir la protección de la estructura de tipos del lenguaje. En contraste, la unión de ALGOL 68 es completamente segura, es decir, la sintaxis del lenguaje es tal que todos los accesos inadecuados se pueden detectar en tiempo de compilación. Debido a que todas las comprobaciones en tiempo de ejecución se especifican explícitamente por el programador, los costes no están ocultos en la implementación.

4. No hay reglas de compatibilidad de tipos especificadas rigurosamente en el "Report" de Pascal, mientras que ALGOL 68 se toma mucho cuidado en definir precisamente la noción de compatibilidad (ver Sección 4.5.2). Como consecuencia, la comprobación de tipos de Pascal reposa frecuentemente en terreno movedizo, y su efecto puede variar en implementaciones distintas.

4.5.2 Compatibilidad de Tipos

Hasta aquí, hemos visto de una forma deliberadamente informal lo concerniente a las reglas de compatibilidad o equivalencia de tipos. Tales reglas nos deberían dar una especificación exacta de cómo deberían aplicarse los mecanismos de comprobación de tipos. Vamos a considerar las siguientes declaraciones Pascal.

```
type t = array[1..20] of integer;
var a,b : array[1..20] of integer;
c : array[1..20] of integer;
d : t;
e,f : record a : integer;
      b : t
    end
```

Es posible definir las dos nociones siguientes de compatibilidad de tipos.

1. Equivalencia de nombre. Dos variables tienen tipos compatibles si tienen el mismo nombre de tipo, predefinido o definido por el usuario, o si aparecen en la misma declaración. Por lo tanto, `a` y `b` (`y` e `y` `f`) tienen tipos compatibles, así como `d`, `e.b` y `f.b`, pero no `a` y `c`. El término "equivalencia de nombre" refleja el hecho de que dos variables que no se han declarado juntas tienen tipos compatibles sólo si su nombre de tipo es el mismo.

2. Equivalencia estructural. Dos variables tienen tipos compatibles si tienen la misma estructura. De acuerdo con esta definición, los nombres de los tipos definidos por el usuario se utilizan precisamente como una abreviación o como un comentario de la estructura que representan, y no introducen ninguna característica semántica nueva. Para verificar la equivalencia estructural, los nombres de los tipos definidos por el usuario se sustituyen por su definición. Este proceso se repite hasta que no queden nombres de tipos definidos por el usuario. Entonces, se considera que los tipos son equivalentes por estructura si tienen exactamente la misma descripción. En el ejemplo anterior, `a,b,c,d,e.b`, y `f.b` tienen tipos compatibles. Esta definición de equivalencia estructural puede conducir a un bucle infinito cuando se utilizan punteros para crear definiciones recursivas de tipos. Los lenguajes que adoptan la equivalencia estructural tienen en cuenta este problema y suministran una regla apropiada (ver Ejercicio 4.19).

En ALGOL 68 se define la compatibilidad de tipos por medio de la equivalencia estructural. Por otra parte, el "Report" de Pascal no especifica el criterio adoptado para la compatibilidad de tipos, y por lo tanto deja que esta importante cuestión se

decida en la implementación.

Una consecuencia desagradable es que un programa aceptado por un compilador puede ser rechazado por otro si las reglas de compatibilidad de tipos de las dos implementaciones difieren.

En la mayoría de los casos, la implementación Pascal original utiliza equivalencia estructural. La elección de la equivalencia de nombre podría hacer ilegal, por ejemplo, asignar un valor entero a una variable que se ha especificado como un subrango de los enteros, a menos que el lenguaje defina unas conversiones apropiadas. La equivalencia de nombre se utiliza para el paso de parámetros.

El concepto de compatibilidad de tipos basado en la equivalencia estructural adoptado por ALGOL 68 va hasta el extremo de ignorar por completo los nombres definidos por el usuario. Por consiguiente, la consistencia de tipos en ALGOL 68 se basa en algo más bien pobre, la noción de tipo puramente sintáctica. Por ejemplo, las declaraciones de tipo en ALGOL 68:

```
mode celsius = int
y
mode fahrenheit = int
```

ocasiona que celsius y fahrenheit sean tipos compatibles. Como consecuencia, el valor de una variable que representa una temperatura en grados celsius puede asignarse correctamente a una variable que representa una temperatura en grados fahrenheit, incluso aunque, presumiblemente, ésta no sea la intención del programador.

La equivalencia de nombre está más cercana que la equivalencia estructural al concepto de tipos abstractos de datos. Si bien no asocia operaciones al tipo, previene de considerar que dos tipos sean compatibles sólo por el hecho de que sus estructuras sean idénticas. Se pueden especificar una serie de propiedades con algo en común utilizando el mismo nombre de tipo, lo cual puede mejorar la legibilidad de los programas. Además, la equivalencia de nombre es más simple de implementar. De hecho, la implementación de la equivalencia estructural requiere un procedimiento de congruencia de patrones que puede ser bastante complicado.

Sin embargo, ambas nociones se pueden ver principalmente como sintácticas más que semánticas. Desde el punto de vista semántico, lo que se debería expresar es la noción de comportamiento idéntico bajo la aplicación de las mismas operaciones, en vez de la compatibilidad de las estructuras representadas; sin embargo, esto requiere que las características del lenguaje soporten la definición de tipos abstractos de datos.

4.5.3 Punteros.

Los punteros empezaron a ser criticados a mediados de los años setenta. Así como los gatos sin restricciones amplían el contexto desde el que se puede ejecutar una sentencia etiquetada, los punteros sin restricciones amplían el contexto desde el que se puede acceder a un objeto.

Los punteros son la herramienta básica suministrada para representar recursivamente objetos definidos. Sin embargo, como tal constructor de lenguaje de bajo nivel, se pueden utilizar para otros propósitos distintos de aquellos para los que originalmente se pensó. Hacen que los programas sean menos comprensibles y frecuentemente inseguros, principalmente por las razones siguientes.

- En algunos lenguajes el uso de punteros puede conducir a serias violaciones de tipo, debido a que los punteros no están cualificados por el tipo del objeto al que pueden apuntar. Por ejemplo, los punteros en PL/I se declaran simplemente como punteros, en lugar de como punteros a un tipo determinado. Las variables declaradas BASED se acceden sólo a través de punteros. Por ejemplo las declaraciones:

```
DECLARE P POINTER,
      X FIXED BASED, /* ENTERO */
      Y FLOAT BASED; /* REAL */
```

declaran P como un puntero, X como un entero, e Y como un real. El acceso a X se hace a través de un puntero, es decir, P --> X indica acceder a un objeto entero X a través de un puntero P. Se hace apuntar un puntero a una variable basada cuando se asigna la variable expresamente, como en:

```
ALLOCATE X SET P;
```

Sin embargo, puesto que P no está cualificado para apuntar únicamente a enteros, se puede además intentar acceder a Y a través del mismo P, como en P --> Y.

En tiempo de traducción, es imposible garantizar que el puntero que se va a suministrar va a apuntar a una variable del tipo correcto. Ya que en algunas implementaciones la comprobación dinámica se considera muy costosa, la solución corriente es que el traductor asuma que el acceso se va a hacer correctamente. Esto puede ocasionar errores en tiempo de ejecución que son muy difíciles de encontrar.

- Un puntero puede resultar peligroso si se queda suelto, es decir, si hace referencia a una posición que tuvo pero que ya no tiene un contenido válido. Un ejemplo clásico de PL/I es el siguiente:

```

BEGIN;
  DCL P POINTER;
  BEGIN; DCL X FIXED; /* ASIGNACION DE MEMORIA
                           PARA X */
    P = ADDR(X); /* P APUNTA AHORA A X */
  END;
  /* EN ESTE PUNTO, X HA DESAPARECIDO PERO */
  /* IP AUN APUNTA A SU POSICION!. EL CULPABLE */
  /* REAL ES LA FUNCION ADDR QUE PERMITE */
  /* QUE SE UTILICE LA DIRECCION DE UNA */
  /* VARIABLE COMO UN VALOR. */
END;

```

Tanto ALGOL 68 como Pascal ligan cada valor de puntero a un tipo de dato específico, y así salvan el problema anterior a. Sin embargo, en términos del problema b., ambos lenguajes tienen un cierto número de inseguridades.

b.1. La restricción de ALGOL 68 de que en una asignación a un puntero, el ámbito del objeto que va a ser apuntado sea al menos tan grande como el del propio puntero, sólo se puede comprobar en tiempo de ejecución. Por ejemplo, vamos a considerar un procedimiento p con dos parámetros formales x, un entero, e y, un puntero a enteros. El que la asignación y := x dentro del procedimiento sea legal, depende de los parámetros reales que obviamente no son conocidos en tiempo de traducción. Como siempre, la comprobación de errores en tiempo de ejecución disminuye la velocidad de ejecución del programa, y la ausencia de dicha comprobación conduce a referencias sueltas no detectadas. Los punteros de Pascal no dan lugar a tales problemas, puesto que sólo se ligan a objetos anónimos ubicados en memoria libre.

b.2. La cantidad de memoria libre asignada durante la ejecución de un programa puede llegar a ser excesivamente grande. No obstante, tan pronto como una zona de la memoria libre deja de ser referenciable, se debería devolver y posteriormente asignarse a nuevas variables de memoria libre. Para hacer esto posible, muchas implementaciones de Pascal suministran el procedimiento estándar dispose, que libera explícitamente memoria. Desgraciadamente, el programador puede pedir la liberación de una variable de memoria libre cuando todavía hay punteros a ella. Por lo tanto se pueden crear referencias sueltas, a menos que se compruebe en tiempo de ejecución la utilización incorrecta del dispose. Tanto ALGOL 68 como SIMULA 67 evitan este problema al no permitir la liberación explícita de variables de memoria libre. En cambio, delegan explícitamente en un recolector de residuos ("garbage collector") que reclama automáticamente la memoria libre no utilizada. El

recolector de residuos se tratará en la Sección 4.7.4.

- c. Los punteros no inicializados, y los evaluados a nil, pueden provocar el acceso incontrolado a la memoria puesto que la cadena de bits encontrada en la posición ligada al puntero se interpretará como un valor de puntero. Para hacer posible la comprobación en tiempo de ejecución, los punteros se deben inicializar automáticamente a nil, siendo el valor de nil una dirección ilegal; consecuentemente, un direccionamiento con un valor nil se puede detectar automáticamente por el hardware en tiempo de ejecución.
- d. Si no se comprueba en tiempo de ejecución la utilización de los registros con variante de Pascal, se podría asignar un entero a un campo bajo una variante y ser interpretado como un puntero bajo otra. Como consecuencia, el programa podría modificar aleatoriamente el contenido de la memoria de una forma totalmente descontrolada. El siguiente ejemplo ilustra este punto.

```

type dañino = record . .
  case indicador : boolean of
    true : (i : integer);
    false: (ref : ^integer)
  end;
var alborotador : dañino;
begin . .
  while b do
    begin alborotador.indicador := true;
      alborotador.i := 000;
      alborotador.indicador := false;
      alborotador.ref := 0;
    end;
  . .
end

```

Como consecuencia de la noción indefinida de compatibilidad de tipos discutida anteriormente, se presenta un problema adicional con los punteros de Pascal. Por ejemplo, después de las declaraciones:

```

type ptr = ^nodo; ref = ^nodo;
  nodo = record unidad : integer;
            siguiente : ptr
  end

```

no está tan claro si es legal asignar una variable de tipo ref, a una variable de tipo ptr, y viceversa.

Euclid resuelve este problema introduciendo las colecciones ("collections"). Las colecciones son unas variables especiales que denotan un conjunto de objetos del mismo tipo. Cuando se declara un puntero, se liga a una colección. Algunas colecciones pueden tener elementos del mismo tipo, y algunos punteros pueden apuntar a la misma colección, pero cada puntero sólo puede referirse a una colección específica. Por ejemplo:

```
type nodos = record . . . end;
var mis_nodos : collection of nodos;
type mi_ref = ^mis_nodos;
var mi_ref1, mi_ref2 : mi_ref;
```

Declaran `mi_ref1` y `mi_ref2` como referencias a una colección de nodos `mis_nodos`. Las variables `mi_ref1` y `mi_ref2` pueden apuntar a nodos de la colección `mis_nodos`, pero otros punteros no pueden hacerlo, a menos que se declaren como ligados a la misma colección. No se definen operaciones para las colecciones; las colecciones solamente se pueden pasar como parámetros, pero nunca se pueden asignar. Esta visión de los punteros tiene una fuerte analogía con los índices de las matrices. Un puntero es un índice dentro de la colección, y la "extracción de valor" en este caso es equivalente a la indexación dentro de la matriz. Como sucede con las matrices, un puntero no puede ser objeto de una "extracción de valor" dentro de un ámbito dado, a menos que su colección sea visible dentro de ese ámbito, es decir, la colección es una variable global o se pasa como un parámetro.

4.5.4 La Estructura de Tipos de Ada

La estructura de tipos de Ada se basa en gran parte en Pascal. No pretendemos dar una visión detallada y completa de la estructura de tipos de Ada, sino más bien mostrar cómo tal estructura supera un gran número de problemas e inseguridades presentadas anteriormente. La discusión de las características de Ada para describir tipos abstractos de datos se hará en la Sección 4.6.2.2.

Ada pone mucho énfasis en distinguir entre las propiedades dinámicas y estáticas de los tipos. Las propiedades estáticas son aquellas que pueden, y deben, comprobarse por un análisis del programa en tiempo de traducción. Las propiedades dinámicas son aquellas que en general, pueden comprobarse solamente en tiempo de ejecución; por ejemplo, restricciones de rango sobre enteros, o de índice sobre matrices. Para aclarar más la distinción, Ada permite al programador especificar una propiedad dinámica sobre un tipo definiendo un subtipo. Por ejemplo,

```
type SABOR is (CHOCOLATE, MENTA, MELOCOTON, FRESA, VAINILLA,
               QUESO, TOMATE, AJO, CEBOLLA);
subtype SABOR_HELADO is SABOR range CHOCOLATE .. VAINILLA;
subtype ENT_CORTO is INTEGER range -10..10;
subtype ENT_CORTO_POSIT is ENT_CORTO range 1..10;
subtype MI_CONJUNTO_ENT is INTEGER range A..B;
```

Una variable de un subtipo tal como `SABOR_HELADO` hereda todas las propiedades del tipo `SABOR`, pero tiene valores que satisfacen una cierta restricción, el pertenecer al subconjunto de `CHOCOLATE` a `VAINILLA`. El último ejemplo (`MI_CONJUNTO_ENT`) muestra que las restricciones pueden involucrar expresiones que no se pueden evaluar estáticamente; más bien, se asume que se van a evaluar cuando se elabore la declaración de subtipo a la entrada del ámbito donde aparece la declaración.

Se emplea una aproximación similar para las matrices que, al contrario que en Pascal, pueden ser dinámicas. El tipo matriz se caracteriza por el tipo de los componentes, el número de índices, y el tipo de cada índice; los valores de los límites no se consideran como parte del tipo de la matriz. Por ejemplo, vamos a considerar la declaración siguiente.

```
type MI_PEDIDO is array(INTEGER range <>) of SABOR_HELADO;
-- El simbolo <> quiere decir rango sin especificar
```

Los objetos del tipo `MI_PEDIDO` tienen componentes del tipo `SABOR_HELADO` indexados por valores de un rango sin especificar de los enteros. Sin embargo, para cualquier objeto de tipo matriz, se deben especificar los límites de cada índice. Por ejemplo, podemos escribir

```
subtype PEDIDO_MENSUAL is MI_PEDIDO(1..31);
subtype PEDIDO_ANUAL is MI_PEDIDO(1..365);
```

y luego declarar

```
PEDIDO_JULIO, PEDIDO_AGOSTO : PEDIDO_MENSUAL;
PEDIDO_ULTIMO_ANO : PEDIDO_ANUAL;
```

Otra forma de fijar los límites es por inicialización. Por ejemplo, podemos declarar

```
PEDIDO_SEPTIEMBRE : constant MI_PEDIDO := PEDIDO_JUNIO;
```

La variable `PEDIDO_SEPTIEMBRE` denota un objeto que sólo se puede leer, debido a la especificación `constant`, que satisface las mismas restricciones de rango y es inicializado con los mismos valores que `PEDIDO_JUNIO`.

Una forma incluso más interesante de fijar los límites es mediante el uso de parámetros. Por ejemplo, la siguiente función recibe un objeto `LISTA` del tipo `MI_PEDIDO` y un objeto `ELECCION` del tipo `SABOR_HELADO`, y evalúa cuántas veces aparece la `ELECCION` en la `LISTA`.

```

function CUANTAS_VECES(LISTA : MI_PEDIDO; ELECCION :
                           SABOR_HELADO) return INTEGER is
  RESULTADO : INTEGER := 0;
begin for I in LISTA'FIRST .. LISTA'LAST loop
  if LISTA(I) = ELECCION
    then RESULTADO := RESULTADO+1;
  end if;
end loop;
return RESULTADO;
end CUANTAS_VECES;

```

La variable RESULTADO es local a CUANTAS_VECES, y se inicializa a cero cuando se declara. Las variables de bucle, por ejemplo I, se consideran declaradas implícitamente. El parámetro formal LISTA tiene automáticamente los mismos límites que el parámetro real, y se puede acceder a tales límites concatenando los nombres de atributo FIRST y LAST al nombre de la matriz. Por consiguiente, se puede llamar a la función con matrices de diferentes tamaños como parámetros. Por ejemplo

```

A := CUANTAS_VECES(PEDIDO_ULTIMO_ANO,VAINILLA) +
     CUANTAS_VECES(PEDIDO_JULIO, VAINILLA);

```

Además, es posible declarar una matriz local al procedimiento, cuyos límites dependen de los parámetros; por ejemplo,

```

PROVISIONAL : MI_PEDIDO(FIRST .. LISTA'LAST);

```

Como en ALGOL 68, las cadenas de caracteres se contemplan como vectores de caracteres y se definen como sigue:

```

subtype NATURAL is INTEGER range 1..INTEGER'LAST;
-- INTEGER'LAST es el mayor entero representable
type CADENA is array (NATURAL range <>) of CHARACTER;

```

Otra característica interesante es cómo define Ada las uniones discriminadas. El ejemplo de Pascal presentado en la Sección 4.3.2.2, en la discusión de los registros con variante, se puede escribir en Ada de la forma siguiente.

```

type DEPARTAMENTO is (DOMESTICO,DEPORTES,DROGUERIA,ALIMENTACION,
                       BEBIDAS);
subtype MES is INTEGER range 1..12;
type ARTICULO(DISPONIBLE : BOOLEAN := TRUE) is
  record
    PRECIO : REAL;
    case DISPONIBLE of
      when TRUE => CANTIDAD : INTEGER;
                           DONDE : DEPARTAMENTO;
      when FALSE => MES_ESPERADO : MES;
    end case;
  end record;

```

El tipo ARTICULO tiene un discriminante, DISPONIBLE, que define las variantes posibles de ARTICULO. El valor inicial por defecto del discriminante se declara arriba a TRUE.

Es posible definir subtipos en los que se congele la variante, por ejemplo

```

subtype FUERA_DE_STOCK is ARTICULO(FALSE);

```

En tal caso, la cantidad de espacio que va a reservar el traductor para las variables del subtipo FUERA_DE_STOCK es exactamente el necesario para la variante, y las variables del subtipo no pueden cambiar su variante.

Las variables de tipo unión discriminada se manejan en Ada de una forma segura. De hecho, el discriminante es obligatorio y no se puede asignar directamente. En ausencia de valores iniciales por defecto para los discriminantes, se debe dar una restricción de discriminante para toda declaración de un objeto, posiblemente suministrando el nombre de un subtipo que incorpore tal restricción. Por ejemplo

```

X : ARTICULO(FALSE);
Y : FUERA_DE_STOCK;

```

El valor de un discriminante solamente se puede cambiar para aquellos objetos que no hayan sido restringidos explícitamente. Y lo que es más, el discriminante sólo se puede cambiar mediante una asignación al registro como un todo, y no por asignación única del discriminante. Esto prohíbe la producción de objetos inconsistentes. Por ejemplo, después de la declaración

```

COCA_COLA : ARTICULO; -- DISPONIBLE tiene el valor inicial
                        -- por defecto TRUE

```

podemos escribir la sentencia siguiente

```

COCA_COLA := Y; -- La variante se pone a FALSE puesto que
                  -- Y es del subtipo FUERA_DE_STOCK

```

o la sentencia

```

COCA_COLA := (PRECIO => 1.99,DISPONIBLE => TRUE,CANTIDAD => 1500,
               DONDE => ALIMENTACION);
-- La parte derecha de la asignación es un valor de
-- registro especificado campo a campo

```

Finalmente, un acceso a un componente tal como

```

COCA_COLA.DONDE

```

se convierte automáticamente por el compilador a la siguiente comprobación en tiempo de ejecución.

```

if not COCA_COLA.DISPONIBLE then raise CONSTRAINT_ERROR
end if;           -- Se alcanza error en tiempo de ejecución
que precede a la manipulación requerida de COCA_COLA.DONDE.

```

La compatibilidad de tipos en Ada se basa en la equivalencia de nombre. Los objetos que pertenecen a subtipos distintos procedentes del mismo tipo son compatibles. Las restricciones se deben comprobar durante la traducción siempre que sea posible, y en los demás casos, en tiempo de ejecución.

Los punteros se definen de una forma similar a la utilizada en Pascal. Además, dos punteros pueden referirse al mismo objeto sólo si sus tipos son compatibles. Utilizando la terminología Euclid, cada tipo de puntero tiene una colección asociada implícitamente. Dos punteros de tipos compatibles se asocian con la misma colección implícita. Se garantiza que dos punteros sin tipos compatibles no apuntan a la misma colección.

La Figura 4.6 muestra la estructura de tipos de Ada.

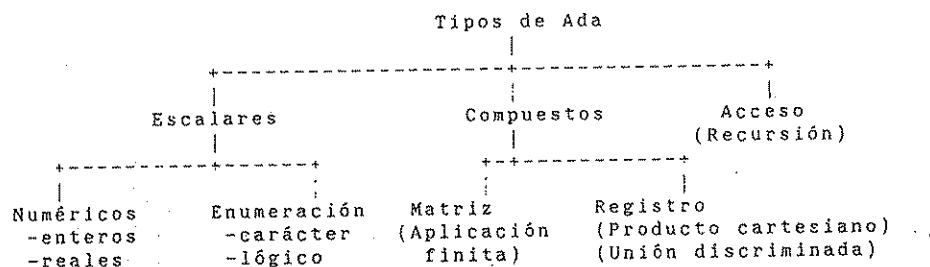


Figura 4.6 Estructura de tipos de Ada.

4.6 TIPOS ABSTRACTOS DE DATOS

El concepto de tipo, tal como aparece definido en Pascal y ALGOL 68, ha sido un paso importante de cara a la consecución de un lenguaje capaz de soportar programación estructurada. Sin embargo, estos lenguajes no soportan totalmente la filosofía del ocultamiento de la información, según la cual los programas se desarrollan por medio de una descomposición del problema basada en el reconocimiento de abstracciones. La abstracción de datos útil para este fin no clasifica los objetos con arreglo a la estructura de su representación, sino que lo hace con arreglo a su comportamiento esperado. Tal comportamiento se puede expresar en términos de las operaciones que están asociadas a esos datos, y estas operaciones son el único medio para crear, modificar y acceder a los objetos.

A partir de SIMULA 67, otros lenguajes como Pascal Concurrente, CLU, Mesa, Euclid, Modula y Ada, permitieron al

programador, en diferente grado unos de otros, definir un tipo abstracto de dato proporcionando construcciones especiales en el propio lenguaje para encapsular la representación y las operaciones concretas que son las que soportan nuestra visión abstracta.

Como ya vimos en la Sección 2.2, la existencia de un mecanismo de encapsulamiento para encerrar en una unidad textual las estructuras de datos que representan los objetos y los procedimientos que representan las operaciones, mejora la localización de las modificaciones. Un cambio en una estructura de datos normalmente implica un cambio en los procedimientos que acceden a ella, pero el efecto de dichos cambios está confinado dentro de las fronteras del mecanismo de encapsulamiento. De igual forma, un cambio en el programa que utiliza la abstracción del dato no tiene ningún efecto sobre la corrección de la parte de programa que está encerrada dentro del mecanismo.

La posibilidad de parametrizar abstracciones definidas por el usuario proporciona una herramienta todavía más flexible y más potente para expresar abstracciones en un lenguaje. Por ejemplo, los tipos abstractos de datos "cola de enteros" y "cola de clientes" pueden ofrecer el mismo comportamiento abstracto, independientemente del tipo (abstracto) de los elementos que estén encolados. Una manera natural de expresar este hecho es definir una "cola" de tipos genéricos (o generador de tipos), donde el tipo de los elementos almacenados es un parámetro.

La Sección 4.6.1 estudia el concepto de class tal como fue propuesto en SIMULA 67 y evalúa su potencia y flexibilidad a la hora de definir tipos abstractos de datos. En la Sección 4.6.2 veremos una justificación y una explicación detallada de una variante del mecanismo class de SIMULA 67, así como una presentación de las facilidades proporcionadas por CLU y Ada.

4.6.1 El Mecanismo "Class" de SIMULA 67

SIMULA 67, un lenguaje de propósito general, fue definido como una extensión de ALGOL 60 para descripción de sistemas y para simulación. La principal diferencia con ALGOL 60 es el concepto de class, el cual, aunque estaba inspirado inicialmente en las necesidades particulares de la simulación discreta, fue posteriormente utilizado como una herramienta general para diseñar programas organizados en niveles de abstracción. Su generalización a la programación concurrente condujo al concepto de monitor de Pascal Concurrente. El mismo concepto fue el origen de la idea de los tipos abstractos de datos, soportada por Pascal Concurrente (con la construcción class) y por CLU (con la construcción cluster); y en general, sugirió cómo ofrecer construcciones del lenguaje para soportar el concepto de ocultamiento de la información.

SIMULA 67 tiene una estructura convencional de anidamientos tipo ALGOL. Una "class" puede declararse en la lista de declaraciones que aparece en la cabecera de un bloque, junto con

procedimientos y variables. Una declaración de "class" tiene la forma general:

```
<cabecera_de_class>;<cuerpo_de_class>
```

<cabecera_de_class> contiene el nombre de la "class" y los parámetros formales. **<cuerpo_de_class>** es un bloque convencional, es decir, puede contener declaraciones de variables, procedimientos y "class" locales, así como sentencias ejecutables.

Por ejemplo, el concepto de números complejos en representación polar se puede describir por la siguiente declaración de "class". Los parámetros **x** e **y** representan las componentes del número complejo en forma cartesiana, y las variables locales **ángulo** y **radio** representan las componentes en forma polar. Las funciones **sqrt**, **abs** y **arctan** son funciones predefinidas por el lenguaje, mientras que **error** (que no está especificado) es un procedimiento accesible desde la "class". La variable global **epsilon** representa un valor real positivo que se utiliza como una aproximación a cero, y la variable global **pi** representa el valor de π .

```
class complejo (x,y); real x,y;
begin
  real angulo, radio;
  radio := sqrt (x ** 2 + y ** 2);
  if abs (x) < epsilon
    then begin if abs (y)<epsilon
      then error
      else begin if y > epsilon
        then angulo := pi/2;
        else angulo := 3*pi/2
      end
    end
  else angulo := arctan(y/x)
end complejo;
```

De la misma forma que en las declaraciones de tipos de ALGOL 68 y Pascal, una declaración de "class" define un prototipo, un original, de una clase de objetos de datos. Cada instancia o copia del prototipo es un objeto real y manipulable.

Estas instancias de "class" (llamadas objetos en terminología SIMULA) se pueden crear dinámicamente en un número arbitrario, y se pueden referenciar individualmente sólo a través de un puntero. Por ejemplo, en el siguiente par de sentencias, **c** se declara como un puntero a un complejo en la primera sentencia, y se hace que apunte a un objeto complejo recién creado en la segunda sentencia.

```
ref (complejo) c;
c := new complejo (1.0,1.0)
```

(":=" es el símbolo de asignación y se lee "denota"). El efecto de dichas sentencias se muestra en la Figura 4.7.

c	
0.78	ángulo
1.42	radio
1.0	x
1.0	y

Figura 4.7 Un objeto complejo.

Los atributos de una instancia de una "class" son instancias de las variables locales declaradas en el cuerpo de la misma y de los parámetros listados en su cabecera. Como los campos de un registro de Pascal, los atributos de una instancia de una "class" son accesibles desde fuera de ella mediante el uso de la notación puntual. Por ejemplo, después de que las sentencias anteriores hayan producido la generación de la instancia de la "class" apuntada por **c**, la ejecución de las sentencias

```
mi_angulo := c.angulo;
mi_radio := c.radio;
mi_x := c.x;
mi_y := c.y;
```

produce los siguientes valores:

```
mi_angulo = 0.78, mi_radio = 1.42, mi_x = 1.0, mi_y = 1.0
```

Las declaraciones de una "class" tienen alguna semejanza con las declaraciones del tipo record de Pascal o de ALGOL 68. SIMULA 67 no tiene realmente un constructor explícito del producto cartesiano, y la "class" se puede usar para ese fin. Sin embargo, a diferencia de ALGOL 68 o Pascal, en SIMULA 67 un acceso sólo se puede hacer a través de variables de referencia, ya que las instancias de una "class" no tienen nombre. Además, cada instancia se debe generar explicitamente utilizando una sentencia new. Finalmente, diremos que una instancia de una "class" es un objeto inicializado, ya que su cuerpo se ejecuta automáticamente con la sentencia new.

Desde un punto de vista más general, las "class" son mecanismos de encapsulamiento que soportan la definición de tipos abstractos de datos. Una "class" puede encerrar los procedimientos que realizan las operaciones sobre los datos. Por ejemplo, los procedimientos suma y multiplicación, para sumar y multiplicar números complejos, se pueden encerrar dentro del cuerpo de la "class" complejo. Esas operaciones también son atributos accesibles a través de la notación puntual, **c.suma**, **c.multiplicacion**, y pueden tener parámetros. Por ejemplo, las cabeceras de los procedimientos suma y multiplicación encerrados dentro de la class complejo son

```

procedure suma (operando); ref (complejo) operando;
y
procedure multiplicacion (operando); ref (complejo) operando;
La suma de los números complejos c1 y c2 se puede expresar como
c1.suma(c2)
es decir, c2 se pasa como parámetro al procedimiento suma
asociado a c1.

```

Sin embargo, esta notación no es adecuada para operaciones binarias sobre variables de un tipo abstracto, ya que la misma operación también se podría haber expresado como

```
c2.suma(c1)
es decir, c1 se pasa como parámetro al procedimiento suma
asociado a c2.
```

Un problema más serio es que la notación puntual proporciona acceso a todos los atributos de un objeto, y así, por ejemplo, no está prohibido escribir

```
c1.angulo := c2.angulo;
c1.radio := c2.radio+c3.radio;
```

para hacer que c1 exprese el resultado de la suma de los números complejos c2 y c3, si se sabe que los ángulos de c2 y c3 son iguales. En otras palabras, suma y multiplicación no son las únicas operaciones que pueden manipular objetos de datos complejos, ya que el acceso directo a la representación está permitido.

SIMULA 67 tiene también una construcción especial que permite al programador especificar uniones discriminadas de tipos, así como tipos abstractos de datos genéricos. Se trata de la "subclass", que tiene de hecho un campo de aplicación más general y se pretende con ella dotar al programador de una herramienta general para la organización de sistemas en niveles de abstracción. Esto se consigue prefijando una "class" con el nombre de otra.

Supongamos que queremos definir la abstracción "pila de elementos" (de tipos posiblemente no homogéneos). Las operaciones sobre una pila deberían permitir al programador referirse al primer elemento (cima), insertar un elemento nuevo en la primera posición (meter), eliminar el primer elemento (sacar), y comprobar si la pila está vacía (vacía). Tales operaciones son independientes de los tipos particulares de elementos que están almacenados en la pila. En primer lugar describiremos la clase de elementos que se pueden apilar.

```

class miembro_de_pila;
begin ref (miembro_de_pila) siguiente_miembro;
    siguiente_miembro:= none
end

```

Esto especifica que la única propiedad compartida por todos los objetos apilables es la existencia de un atributo que es una referencia al siguiente elemento de la pila. Cada instancia de miembro_de_pila tiene el atributo siguiente_miembro inicializado a none, el valor de un puntero nulo. La clase pila se puede describir separadamente, especificando únicamente las operaciones aplicables a todos los objetos almacenables.

```

class pila;
begin ref (miembro_de_pila) primero;
ref (miembro_de_pila) procedure cima;
    cima:= primero;
procedure sacar;
if !=vacia then primero:= primero.siguiente_miembro;
procedure meter (e); ref (miembro_de_pila) e;
begin if primero == none
    then e.siguiente_miembro := primero;
    primero := e;
end meter;
boolean procedure vacia;
    vacia := primero == none;
    primero := none
end pila;

```

El procedimiento cima devuelve el elemento más alto, es decir, un ref(miembro_de_pila) de una pila no vacía. El símbolo "!" en el procedimiento sacar significa "no". El símbolo "==" en el procedimiento meter significa "no igual". El procedimiento vacia devuelve un valor boolean; el símbolo "==" significa "igual". Cada instancia de una "class" pila se inicializa como una pila vacía, ya que su atributo primero se iguala a none.

Una vez definidas dos "class", miembro_de_pila y pila, podemos ahora crear objetos apilables de un tipo particular, por ejemplo complejo, prefijando la "class" complejo.

```
miembro_de_pila class complejo (...)
```

```
end complejo
```

especifica que los objetos generados por

```
new complejo
```

tienen todos los atributos de miembro_de_pila, así como los atributos de complejo. En otras palabras, son números complejos que se pueden apilar. complejo es una subclase de miembro_de_pila, y está más especializada que miembro_de_pila.

Si declaramos `p` de tipo `ref (pila)`, podemos crear una pila de números complejos haciendo

```
p:= new pila;
```

La pila `p` ahora no contiene ningún elemento, por tanto `p.vacia` devuelve `true`. Si tenemos varios objetos complejos, `c1, c2, c3`, podemos introducirlos en la pila de la siguiente manera

```
p.meter(c1);
p.meter(c2);
p.meter(c3);
```

Podemos ahora mirar el elemento superior por medio de `p.cima`, y podemos quitar el elemento superior mediante `p.sacar`.

Hay que hacer notar que dado que primero es accesible desde el exterior, no existe ninguna protección que impida a un usuario modificar accidentalmente `p.primer` y perder así la pila completa (véase el Ejercicio 4.10).

La "class" `miembro_de_pila` puede verse como una unión de los tipos correspondientes a todas sus subclases. Por ejemplo, si en el programa aparece la siguiente definición

```
miembro_de_pila class vector (...)
```

```
end vector;
```

una variable de tipo `ref (miembro_de_pila)` puede referirse a la vez a un `miembro_de_pila` complejo y a un `miembro_de_pila` vector.

De hecho, los objetos vector y los objetos complejo se pueden introducir en la misma pila. Lo único que la pila comprueba es que existe un atributo `siguiente_miembro` del tipo `ref (miembro_de_pila)` en el objeto que ella manipula. Todas las subclases de `miembro_de_pila` incluyen tal atributo.

Esta libertad también origina inseguridades. Por ejemplo, si introducimos dos objetos vector, `v1` y `v2`, en la pila anterior, es decir, `p.meter(v1);p.meter(v2);`, una activación posterior de `sacar` puede proporcionar bien un objeto vector o bien un objeto complejo. Es posible por tanto intentar acceder a un atributo en un objeto que no tiene tal atributo, por ejemplo, `v.angulo`. Aunque SIMULA 67 ofrece la posibilidad de examinar el tipo de un objeto (`if v is vector then ...`) no se obliga al programador a hacerlo. En otras palabras, a pesar de que el objetivo es definir la pila como un tipo genérico, se debe tener cuidado de no introducir miembros de tipos distintos en la misma instancia de pila.

Otra característica interesante de la "class" de SIMULA es la posibilidad de crear instancias de "class" que trabajan de una

forma quasi-paralela, es decir, intercalada. De hecho, los cuerpos de las "class" se pueden activar como corrutinas. Este tema se trata en el Capítulo 5.

En resumen, la "class" de SIMULA 67 combina varios conceptos interesantes:

- a. Permite la agrupación de objetos relacionados entre sí por algún aspecto de la programación, en una construcción de encapsulamiento.
- b. Proporciona una útil construcción (subclase) para la descomposición jerárquica del sistema.
- c. Ve los objetos abstractos como entidades a las que sólo se puede acceder a través de una "referencia", y proporciona facilidades explícitas para manejar referencias.
- d. Permite la especificación de una ejecución quasi-concurrente de las instancias de las "class".

4.6.2 Abstracción y Protección: CLU y ADA

El diseño inicial de SIMULA 67 no preveía ninguna forma de protección sobre las "class", cuyos atributos podían ser, quizás involuntariamente, manipulados desde fuera de las mismas.

La invisibilidad de la representación de los tipos abstractos de datos es una característica importante de los lenguajes de programación más recientes como Pascal Concurrente y CLU. En otros lenguajes posteriores, como Mesa, Euclid, Modula y Ada, la construcción que suministran para encapsulamiento no se limita a esconder la implementación del tipo abstracto, sino que actúa como un filtro para el que es posible controlar explícitamente qué detalles internos de la unidad encapsulada son visibles desde el exterior (es decir, son exportados). En algunos casos, también es posible especificar qué información externa es visible desde el interior del módulo (es decir, importada por el módulo). Esta visión de los módulos como dispositivos ocultadores de la información proporciona una poderosa herramienta del lenguaje para la estructuración de sistemas, que es especialmente válida para programas grandes y complejos. Las Secciones 4.6.2.1 y 4.6.2.2 revisan las construcciones ofrecidas por CLU y Ada para describir abstracciones de datos.

4.6.2.1 CLU

CLU permite definir tipos abstractos de datos a través de la construcción "cluster", la cual, junto con otras características del lenguaje, vamos a presentar con un ejemplo.

Queremos definir el tipo abstracto de datos número complejo, con las siguientes operaciones:

- crear: recibe un par de números reales como parámetros y genera un número complejo que tiene los dos parámetros como partes real e imaginaria respectivamente (los objetos en CLU deben ser creados explícitamente).
- suma: recibe un par de números complejos como parámetros y entrega el resultado de su suma.
- igual: recibe un par de números complejos como parámetros, y si son iguales entrega un resultado cierto; si no lo son, entrega un resultado falso.

A continuación se describe el "cluster" que representa el tipo abstracto de datos.

```

complejo = cluster is crear, suma, igual
  rep = record [x,y:real]
    crear = proc (a,b:real) returns (cvt)
      return (rep $ (x:a,y:b))
    end crear
    suma = proc (a,b:cvt) returns (cvt)
      return (rep $ (x:a.x + b.x,y:a.y + b.y))
    end suma
    igual = proc (a,b:cvt) returns (bool)
      return (a.x = b.x and a.y = b.y)
    end igual
end complejo

```

La cabecera del "cluster" (cluster is...) lista las operaciones posibles sobre el tipo de dato complejo. La estructura de datos elegida para dar una representación concreta a los objetos del tipo abstracto se especifica con la cláusula rep. En el ejemplo, es un registro con dos campos, uno para la parte real y otro para la parte imaginaria. Los procedimientos que realizan las operaciones, así como otros posibles procedimientos locales no listados en la cabecera, siguen a la cláusula rep. En el ejemplo, son los procedimientos crear, suma e igual.

La palabra reservada cvt, que sólo se puede usar dentro de un "cluster", denota un cambio de punto de vista. Por ejemplo, en el procedimiento suma, los parámetros a y b son de tipo complejo cuando se ven desde fuera del "cluster", pero "son convertidos" a su tipo de representación interna (en el ejemplo, record...) dentro del procedimiento suma. Similarmente, el objeto a devolver, que contiene el resultado de la suma en representación interna, "es convertido" al tipo abstracto complejo a la salida del procedimiento. El significado de return(rep \$...) es que lo que se devuelve es un objeto del tipo de representación interna (es decir, un record); el valor almacenado en el registro se especifica entre llaves usando una notación bastante autoexplicativa. La presencia de cvt en la cláusula returns implica un cambio en la representación abstracta después de la vuelta al programa llamante.

Una característica semántica importante que hace de CLU algo especial es la forma en que el lenguaje contempla las variables y los objetos. En primer lugar, el ámbito y el tiempo de vida son características disjuntas, esto es, las variables se declaran dentro de unidades como en

p:complejo

pero se crean explícitamente como en

p := complejo\$crear(h,k)

El símbolo "\$" es similar a la notación puntual de SIMULA 67. En la sentencia anterior, se llama a la operación crear perteneciente al "cluster" complejo con h y k como parámetros reales. La sentencia return ejecutada por el procedimiento crear genera un objeto (un número complejo cuyas partes real e imaginaria son respectivamente h y k). El objeto devuelto es asignado a p.

En segundo lugar, las variables en CLU son vistas uniformemente como referencias a objetos de datos, y una asignación como

x := e

quiere decir que x va a referirse al objeto que resulte de la evaluación de e. Por tanto, la asignación no modifica el objeto referenciado por x, sino que hace que x pase a referirse a un objeto diferente y el objeto original permanece sin modificación alguna. Este objeto original puede ser referenciado también por otras variables, o puede llegar a ser inaccesible si no existen esas otras variables. A esta forma de asignación se le llama asignación por partición.

El paso de parámetros en CLU se define en términos de asignaciones. Por ejemplo, la llamada

...complejo\$suma (x,y)...

asigna x e y a los parámetros formales a y b respectivamente, los cuales actúan como variables locales. Por tanto, el mismo objeto está compartido por x y a, y por y y b.

Los procedimientos y los "cluster" en CLU pueden ser genéricos, es decir, pueden ser parametrizables por un tipo. Un procedimiento o un "cluster" genérico debe declarar explícitamente qué procedimientos proporciona con arreglo al tipo utilizado como parámetro. Por ejemplo, un "cluster" que trabaje con conjuntos de componentes de tipo t debería tener la siguiente cabecera:

```

conjunto = cluster[t:type] is crea, inserta, borra, pertenece
  where t has igual: proctype (t,t) returns (bool)

```

Esto significa que el tipo conjunto está caracterizado por las operaciones crea, inserta, borra y pertenece; el tipo t debe tener una operación igual, que devuelve un valor lógico cuando es aplicada a dos parámetros de tipo t. Esta información acerca de los parámetros de tipo type permite al mecanismo de comprobación de tipos verificar que las operaciones mencionadas previamente son las únicas que actúan sobre los objetos del tipo utilizado como parámetro dentro del "cluster".

Una declaración de una variable conjunto debe especificar, como un parámetro, el tipo de los componentes. Por ejemplo,

```
e: conjunto [integer]
l: conjunto [bool]
```

De igual forma, las operaciones sobre conjuntos se deben escribir como

```
conjunto [integer]$crea (...)
```

```
conjunto [bool]$crea (...)
```

etc.

4.6.2.2. Ada

El mecanismo de encapsulamiento de Ada es el "package". Una unidad de programa tiene una estructura tipo ALGOL. Como en Pascal, el anidamiento se consigue a través de declaraciones, es decir, una declaración de un subprograma o de un "package" puede contener una declaración de variables, procedimientos, o "packages" locales. Las tareas constituyen otro tipo de unidades que se pueden anidar jerárquicamente (ver Sección 5.2.4.3). El anidamiento también se puede conseguir definiendo nuevos bloques en secuencias de sentencias igual que en ALGOL 60.

Los "packages" pueden ser utilizados para objetivos muy diversos, que van desde la declaración de un conjunto de entidades comunes (variables, constantes, tipos), a la agrupación de un conjunto de subprogramas relacionados entre sí (p. ej., un "package" matemático para la solución de ecuaciones diferenciales), o la descripción de tipos abstractos de datos.

Un ejemplo del primer caso es el siguiente

```
package NUMEROS_COMPLEJOS is
    type COMPLEJO is
        record
            RE: INTEGER;
            IM: INTEGER;
        end record;
    TABLA: array (1..500) of COMPLEJO;
end NUMEROS_COMPLEJOS;
```

La declaración anterior se procesa como si fuera la declaración de las variables y de los tipos encerrados en ella;

por tanto, dichas variables y tipos tienen el mismo ámbito y tiempo de vida que las variables y los tipos declarados en la parte de declaraciones donde aparece la declaración del "package" NUMEROS_COMPLEJOS. Los nombres declarados dentro del "package" se pueden utilizar dentro del ámbito del "package" utilizando la notación puntual (tipo SIMULA 67), como en

```
NUMEROS_COMPLEJOS.TABLA(K)
```

Cuando se trata de agrupar un conjunto de subprogramas relacionados entre sí, o de describir un tipo abstracto de datos, a menudo es necesario esconder algunas entidades locales dentro del "package". En el caso del conjunto de subprogramas, las entidades locales podrían ser variables locales y/o procedimientos locales; en el caso de los tipos abstractos de datos, podría ser alguna representación concreta y quizás algunas variables y procedimientos internos auxiliares. La ausencia de esta facilidad es, como ya hemos visto, un punto débil en SIMULA 67. Ada ha conseguido superar el problema de una forma bastante elegante.

La estructura general de un "package" está compuesta de dos partes: la especificación del "package" y el cuerpo del "package". La especificación contiene exactamente toda la información que es exportada por el módulo, mientras que el cuerpo contiene todos los detalles escondidos de la implementación, y una sección de inicialización que se ejecuta en la activación de la unidad que contiene la declaración del "package".

Como un primer ejemplo, veamos un "package" que exporta las dos funciones siguientes:

- PERTENECE_A(X): da un resultado booleano indicando si X pertenece o no a un cierto conjunto ordenado de enteros;
- POSICION(X): da la posición ordinal de X en el conjunto.

La estructura y el contenido del conjunto ordenado de enteros se esconden dentro del "package"; las operaciones PERTENECE_A y POSICION son las únicas manipulaciones disponibles sobre el conjunto de enteros.

```
package CONJ_ENT is
    function PERTENECE_A (X:INTEGER) return BOOLEAN;
    function POSICION (X:INTEGER) return INTEGER;
end CONJ_ENT;
```

```

package body CONJ_ENT is
    ALMACEN_ENT: array (1..10) of INTEGER;
    function PERTENECE_A (X:INTEGER) return BOOLEAN;
    .
    .
    end PERTENECE_A;
    function POSICION (X:INTEGER) return INTEGER;
    .
    .
    end POSICION;
    -- Ahora viene la inicialización de la matriz
    -- ALMACEN_ENT, es decir, la inicialización del
    -- conjunto.
    .
    .
end CONJ_ENT;

```

Como segundo ejemplo, el siguiente "package" define el tipo abstracto de datos número complejo que ya hemos visto en CLU.

```

package NUMEROS_COMPLEJOS is
    type COMPLEJO is private;
    procedure INICIALIZA (A,B: in REAL; X: out COMPLEJO);
    function SUMA (A,B: in COMPLEJO) return COMPLEJO;
private
    type COMPLEJO is
        record R,I: REAL;
        end record;
end NUMEROS_COMPLEJOS;

package body NUMEROS_COMPLEJOS is
    procedure INICIALIZA (A,B: in REAL; X: out COMPLEJO) is
    begin X.R := A;
        X.I := B;
    end INICIALIZA;

    function SUMA (A,B: in COMPLEJO) return COMPLEJO is
    TEMP: COMPLEJO;
    begin TEMP.R := A.R + B.R;
        TEMP.I := A.I + B.I;
        return TEMP;
    end SUMA;
end NUMEROS_COMPLEJOS;

```

En el ejemplo, el tipo COMPLEJO exportado por el módulo es privado (*private*), es decir, los detalles de la representación encerrados dentro de las parte *private* ... *end NUMEROS_COMPLEJOS;*

de la especificación del "package" no son visibles fuera del mismo. Las variables de tipo COMPLEJO sólo pueden ser manipuladas utilizando los subprogramas INICIALIZA y SUMA exportados por el "package". También están permitidas las operaciones predefinidas de asignación y comprobación de igualdad/desigualdad.

Las variables exportadas también se pueden definir como privadas y limitadas (*limited private*). En este caso, las asignaciones y las comprobaciones de igualdad/desigualdad no estarian definidas automáticamente para el tipo. Si fueran necesarias, se deberian proporcionar explícitamente por el "package" como procedimientos adicionales.

Una diferencia con CLU es que el "package" de Ada no tiene por qué proporcionar una operación explícita crear, ya que la creación se realiza automáticamente cuando se activan las unidades que declaran variables de tipo COMPLEJO. No obstante, el "package" NUMEROS_COMPLEJOS proporciona un procedimiento explícito para la inicialización.

En el ejemplo, los parámetros de los procedimientos se especifican como *in* o *out*. La especificación *in* indica un parámetro de entrada no modificable. La especificación *out* indica un parámetro de salida para el cual el procedimiento genera un valor. El paso de parámetros en Ada se discute en la Sección 5.2.1.1.

Los "packages" (y los procedimientos) pueden ser genéricos (*generic*), en cuyo caso deben crearse antes de poder ser utilizados, de la forma que se indica más adelante. Por ejemplo, un "package" que describa conjuntos de una cardinalidad máxima predefinida, puede tener la siguiente parte de especificación.

```

generic
    type COMPONENTE is private
package MANIPULACION_DE_CONJUNTOS is
    type CONJUNTO is limited private
    procedure INSERTAR (C: in out CONJUNTO; ELEM: in COMPONENTE);
    procedure BORRAR (C: in out CONJUNTO; ELEM: in COMPONENTE);
    function PERTENECE (C: in CONJUNTO; ELEM: in COMPONENTE)
        return BOOLEAN;
private
    type CONJUNTO is
        record ALMACEN: array (1..MAX_CARDINALIDAD)
            of COMPONENTE;
            CARDINALIDAD: INTEGER range 0..MAX_CARDINALIDAD := 0;
        end record;
end MANIPULACION_DE_CONJUNTOS;

```

El procedimiento INSERTAR (o BORRAR) puede ser utilizado para introducir un nuevo elemento en (o hacer desaparecer un

elemento existente de) un conjunto. La función PERTENECE devuelve un valor booleano que es cierto si un elemento ELEM pertenece al conjunto, y falso en caso contrario. La estructura de datos utilizada para representar conjuntos incluye una matriz cuyos componentes son del tipo especificado en la cláusula generic que prefija el "package". El tamaño de los conjuntos viene dado por el valor de la variable global MAX_CARDINALIDAD.

El tipo COMPONENTE es privado (private); es decir, las únicas operaciones permitidas sobre componentes dentro del "package" son asignaciones y comprobaciones de igualdad/desigualdad. Si son necesarias más operaciones, se pueden proporcionar como parámetros genéricos adicionales. El campo CARDINALIDAD, que se utiliza para registrar el número de elementos que están almacenados en un conjunto, se inicializa a cero.

La creación (o encarnación) de un módulo genérico requiere una especificación de los parámetros genéricos reales. Por ejemplo,

```
package ENTEROS is new MANIPULACION_DE_CONJUNTOS (INTEGER);
package SABORES is new MANIPULACION_DE_CONJUNTOS (SABOR);
```

donde el tipo SABOR es el definido en la Sección 4.5.4. Desde un punto de vista semántico, las dos creaciones pueden verse como las declaraciones de dos "packages" distintos, pero que tienen idéntica estructura interna (y por tanto están descritas por el mismo módulo genérico). En el ámbito de estas creaciones, podemos escribir las declaraciones

```
A,B: ENTEROS.CONJUNTO      -- A y B son del tipo CONJUNTO
                                -- creado por ENTEROS.
```

```
C : SABORES.CONJUNTO      -- C es del tipo CONJUNTO
                                -- creado por SABORES.
```

4.7 MODELOS DE IMPLEMENTACIÓN

Esta sección revisa los modelos básicos de implementación de los objetos. La descripción trata de ser independiente del lenguaje, pero los ejemplos que se dan, y el énfasis de la discusión, están orientados a Pascal. Nuestros modelos de representación no tratan de proporcionar una descripción detallada de técnicas eficientes para la representación de objetos dentro de un ordenador, pues ello dependería en gran medida de la estructura hardware del mismo. En lugar de ello, presentaremos soluciones directas junto con algunos comentarios sobre representaciones alternativas más eficientes.

Siguiendo la discusión del Capítulo 3, los datos estarán representados por un par consistente en un descriptor y un objeto. En lugar de utilizar una estructura de datos concreta, describiremos el descriptor abstractamente como un conjunto de

atributos del objeto. La principal razón para ello es que los descriptores normalmente se mantienen en una tabla durante la traducción y, como ya hemos visto, sólo se necesita mantener en tiempo de ejecución un subconjunto de los atributos allí almacenados. Por tanto, el formato de los descriptores, número de campos, formato libre o formato fijo, etc., es fuertemente dependiente de la estructura de la tabla, y el número de atributos almacenados en un descriptor puede variar de tiempo de traducción a tiempo de ejecución.

4.7.1 Tipos Predefinidos y Tipos No Estructurados Definidos por el Usuario

Los enteros y los reales están directamente soportados por el hardware en la mayoría de los ordenadores convencionales que proporcionan aritmética en coma fija y en coma flotante. Las variables enteras y las reales se representan a menudo como se muestra en las Figuras 4.8 y 4.9.

	DESCRIPCIÓN	OBJETO
		Bit de signo
Tipo	Entero	v
Referencia	----->	+-----+ de memoria

Cadena de bits de longitud fija que representan un número (p. ej., en compl. a 2).

Figura 4.8 Representación de una variable entera.

Los valores en un subrango se pueden representar como si fueran del tipo base, de tal forma que la transferencia de valores entre el subrango y el tipo básico no requiera ninguna conversión, únicamente comprobaciones en tiempo de ejecución. El descriptor debe contener los valores de los límites del subrango; dichos valores son necesarios en tiempo de ejecución para realizar las comprobaciones de violación de límites.

Los valores de un tipo por enumeración t pueden corresponder a los valores enteros 0 a n-1, siendo n la cardinalidad de t. Esta representación no introduce ninguna posibilidad de mezclar los valores del tipo t con los valores de cualquier otro tipo por enumeración, si todos los accesos en tiempo de ejecución se hacen a través de un descriptor que contenga la información del tipo. Evidentemente el uso de descriptores es necesario en los lenguajes con tipos.

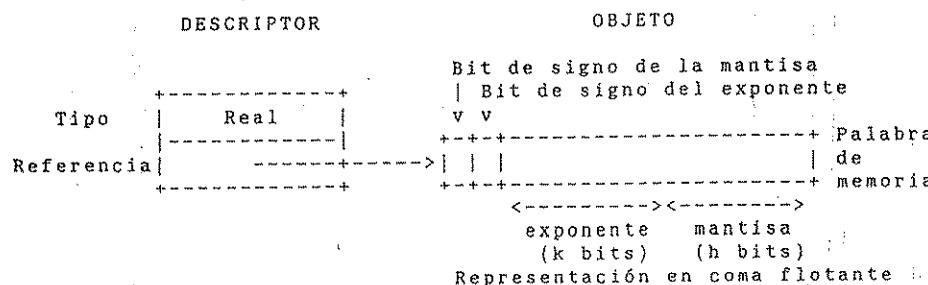


Figura 4.9 Representación de una variable real.

Los tipos lógicos" (booleanos) y los caracteres pueden considerarse tipos por enumeración e implementarse como tales. Para ahorrar espacio, los caracteres se pueden almacenar en unidades más pequeñas que una palabra (p. ej., bytes) si dichas unidades se pueden direccionar directamente por el hardware. También es posible empaquetar en la misma palabra (o byte) varias variables lógicas declaradas en la misma unidad, estando cada una de ellas representada por un bit en concreto. En este caso, el acceso a una variable individual (es decir, a un bit) dentro del grupo (una palabra o un byte) sería menos eficiente sobre máquinas que no tengan direccionamiento directo de bits.

4.7.2 Tipos Estructurados

4.7.2.1 Producto Cartesiano

La representación normal de un objeto del tipo producto cartesiano es una disposición secuencial de los componentes. El descriptor contiene el nombre del tipo producto cartesiano y un conjunto de tripletas (nombre del selector, tipo del campo y referencia al objeto), una para cada campo.

La Figura 4.10 ilustra esta representación para una variable del tipo de Pascal

```
type t = record a: real;
           b: integer
         end
```

Cada componente del producto cartesiano ocupa un número fijo de unidades de almacenamiento direccionables (p. ej., palabras), y basta con dar el nombre del campo dentro del programa para referirse a dicha componente. Los nombres de los campos no pueden estar contenidos en variables. Por tanto, en un lenguaje con chequeo de tipos, no hace falta guardar los descriptores en tiempo de ejecución, y las referencias a cada campo dentro del registro de activación de la unidad en la que el producto

cartesiano es local, pueden ser evaluadas por el traductor. (*)

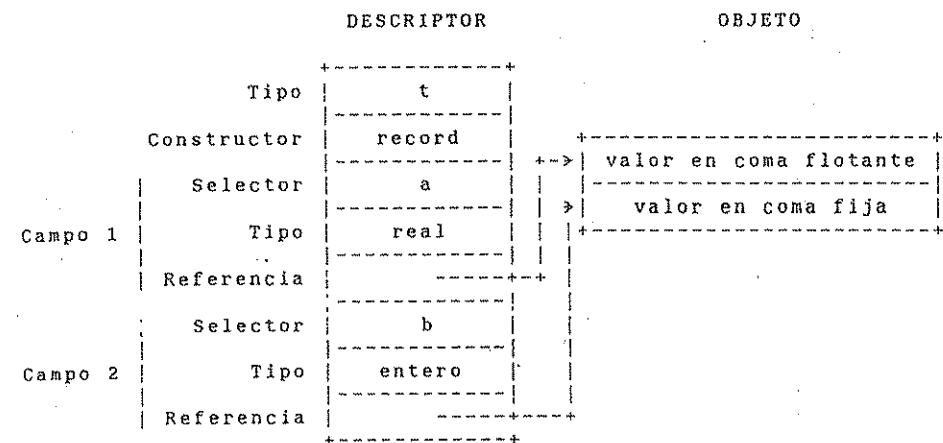


Figura 4.10 Representación de un "record" en Pascal.

En un lenguaje como Pascal, el usuario puede indicar al compilador que gestione el almacenamiento en memoria de una manera eficiente utilizando la opción packed. En este modo de trabajo, cada unidad de almacenamiento se utiliza para guardar más de un componente. Utilizando esta opción, el usuario ahorra memoria a costa de tiempo de proceso.

Por ejemplo, variables del tipo

```
packed record a: char;
           b: 0..7
         end
```

se podrían representar en dos bytes de una única palabra.

4.7.2.2 Aplicación Finita

En el método normal de implementación de una aplicación finita, se reserva un número entero de unidades direccionables de almacenamiento (p. ej., palabras) para cada componente. El descriptor contiene el nombre del tipo de aplicación finita, el nombre del tipo base del tipo de dominio junto con los valores de

(*) Esto es cierto en los registros de Pascal. Por razones de simplicidad, no contemplamos aquí la posibilidad de tener matrices dinámicas como componentes de un producto cartesiano (como en ALGOL 68).

los límites, el nombre del tipo imagen junto con el número de posiciones necesarias para almacenar cada elemento, y la referencia a la primera posición del área donde se almacena el objeto. Por ejemplo, la declaración de Pascal

```
type a = array [0..10] of real
```

se podría representar como indica la Figura 4.11.

	DESCRIPCIÓN	OBJETO
Tipo	a	+> valor en coma flotante 0
Constructor	matriz	" "
Tipo base	entero	" "
indice	Límite inferior	0
	Límite superior	10
Tipo de los componentes	real	" "
No. de posiciones	1	" "
Referencia	" "	10

Figura 4.11 Representación de una matriz en Pascal.

El acceso a los elementos dentro de la matriz se puede hacer utilizando una variable que haga de subíndice, por tanto, los límites superior e inferior se deben guardar en un descriptor de tiempo de ejecución con el fin de realizar comprobaciones de violación de límites en ejecución.

Una referencia a $a[i]$ se calcula como un desplazamiento a partir de la dirección (b) del primer componente de la matriz (dentro del registro de activación de la unidad en la que la matriz es local). Si el tipo dominio es un subrangó $m..n$ y el número de palabras ocupadas por cada elemento es k , el desplazamiento necesario para acceder a $a[i]$ sería $k(i-m)$. Por tanto, una referencia a $a[i]$ se podría expresar como $b+k(i-m) = (b-km)+ki = b'+ki$, donde b' es una constante que se puede calcular en tiempo de compilación.

De igual forma que en la discusión presentada en la Sección 3.6.2, en un lenguaje que soporte matrices dinámicas el descriptor se puede dividir en una parte estática y una parte dinámica. La primera contiene la información que se utiliza únicamente en tiempo de compilación (como el tipo de los componentes de la matriz), y una referencia a la parte dinámica.

La parte dinámica se ubica en tiempo de ejecución dentro del registro de activación de la unidad que declara la matriz, con un desplazamiento conocido en tiempo de compilación. Contiene una referencia al objeto matriz (que en general sólo podrá ser evaluada en tiempo de ejecución), y los valores de los límites. Cualquier acceso a una matriz dinámica ha de realizarse a través del descriptor dinámico, llamado vector de información.

4.7.2.3. Secuencias

Tanto las secuencias de caracteres como las secuencias de registros en un dispositivo de almacenamiento masivo se pueden representar de muchas maneras, en función, tanto de la semántica del lenguaje, como de ciertas características de la arquitectura de la máquina, esto último especialmente en el caso de ficheros.

En Pascal, las cadenas de caracteres son simplemente matrices unidimensionales de caracteres; por tanto, sus longitudes se fijan estáticamente y no se pueden cambiar. En Ada, las cadenas de caracteres son matrices unidimensionales dinámicas de caracteres; por tanto, sus longitudes no son conocidas normalmente en tiempo de compilación, y sólo llegan a determinarse a la entrada de la unidad en la que se declara la variable de que se trate. En los dos lenguajes las cadenas de caracteres se pueden representar como cualquier otra matriz (Sección 4.7.2.2).

En otros lenguajes, como SNOBOL4 y ALGOL 68, las cadenas de caracteres pueden variar de longitud arbitrariamente y no tienen por tanto límite superior especificado por el usuario. Como hemos visto, las cadenas de este tipo son variables dinámicas y deben ser ubicadas en una zona de memoria libre. La Figura 4.12 muestra un ejemplo de una posible representación para una cadena de longitud 5. El descriptor se divide en una parte estática y una parte dinámica; esta última se ubica en la pila y contiene la longitud actual de la cadena (útil para comprobaciones dinámicas), y una referencia a la cabeza de la cadena. La cadena se ubica en la zona de memoria libre como una lista encadenada de palabras, cada una de las cuales puede contener uno o más caracteres. El número de caracteres almacenados en una palabra depende del tamaño de esta última; la Figura 4.12 supone que cada palabra contiene dos caracteres.

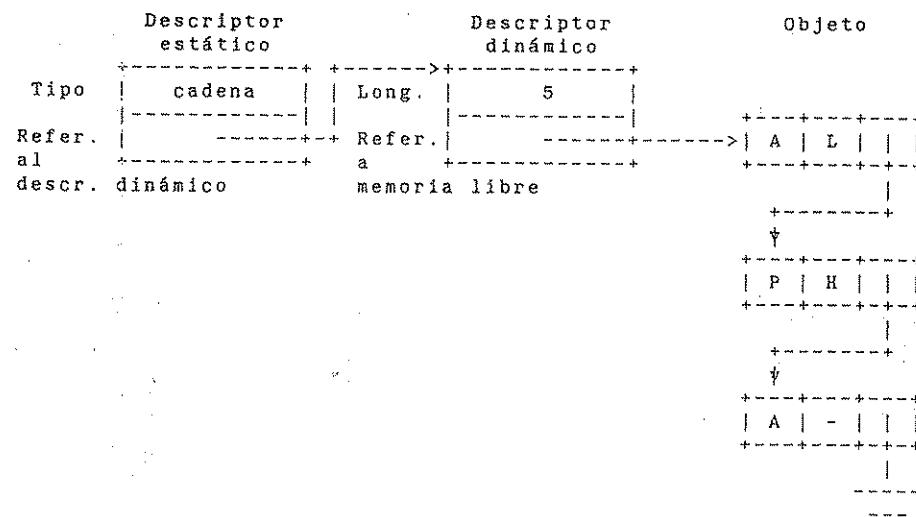


Figura 4.12 Representación de una cadena de caracteres de longitud variable.

4.7.2.4 Unión Discriminada

Una variable perteneciente a un tipo unión discriminada no se ajusta a una variante fija, sino que puede variar como consecuencia de asignaciones. Por tanto, la cantidad de espacio necesario para ubicar la variable debe ser el suficiente para almacenar los valores de la variante que requiera más espacio.

Ada proporciona la opción de ajustar una variable a una variante específica. En este caso, la cantidad de espacio necesario puede ser exactamente la requerida por la variante.

La Figura 4.13 muestra la representación de una variable del tipo unión discriminada.

```
type v = record a: integer;
      case b: boolean of
        true: (c:integer);
        false:(d:integer;
                  e:real)
      end
```

El campo indicador tiene una entrada en el descriptor que apunta a una tabla de selección. Para cada posible valor del indicador, la tabla de selección contiene una referencia a un descriptor de la variante asociada.

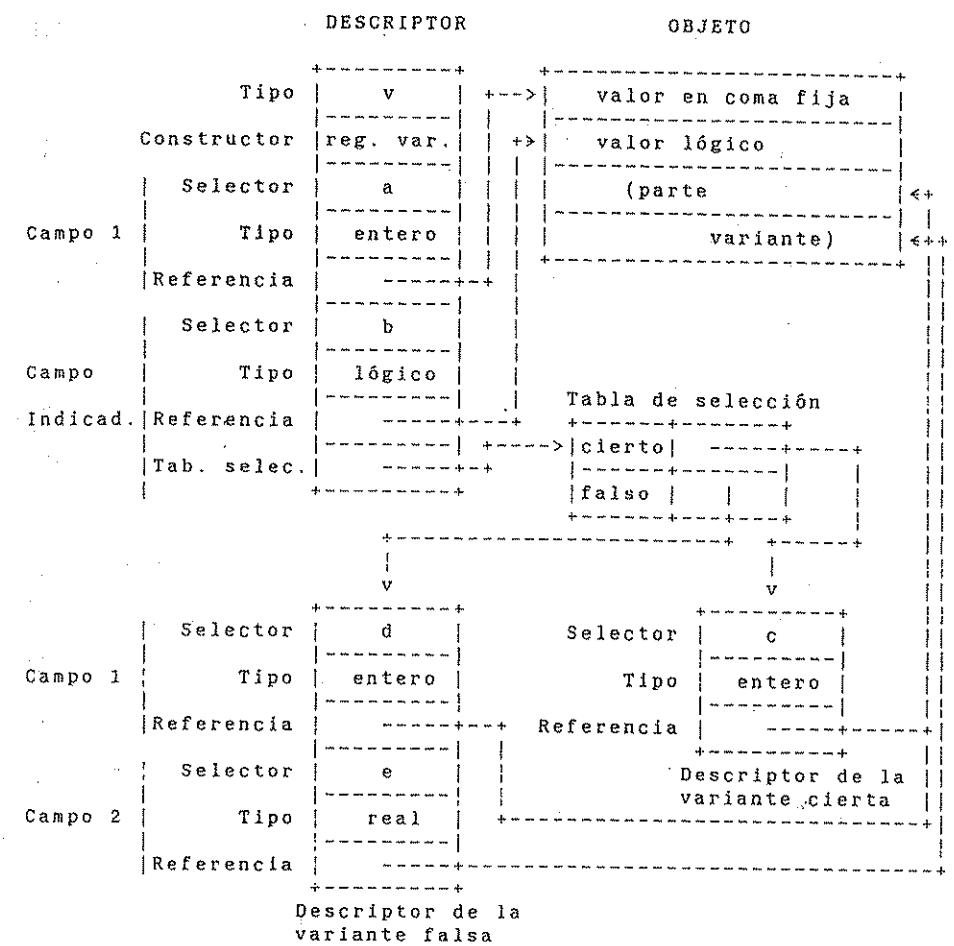


Figura 4.13 Representación de una unión discriminada.

4.7.2.5 Conjunto Potencia

Es posible implementar eficientemente los conjuntos potencia, tanto en términos de tiempo de acceso y de manipulación como en cantidad de almacenamiento necesario, siempre sobre la hipótesis de que una palabra de la máquina tenga al menos tantos bits como miembros pueda haber en un conjunto. La presencia del elemento i-ésimo en el tipo base en un cierto conjunto C se indica por la presencia de un "1" como el valor del bit i-ésimo de la palabra asociada a C. El conjunto vacío se representa por

una palabra con todos los bits a cero. La unión entre dos conjuntos se realiza fácilmente mediante la operación or entre las dos palabras asociadas, y la intersección, mediante la operación and. Si la máquina no permite acceso a nivel de bit, la comprobación de pertenencia implica un desplazamiento del bit en cuestión a un lugar accesible, que puede ser el bit de signo de la palabra, o la utilización de una máscara.

La existencia de una representación de este tipo es la razón de la imposición de un límite máximo en la cardinalidad de los conjuntos, que normalmente es el tamaño de una palabra de memoria.

4.7.2.6 Punteros

Una variable puntero tiene un valor que es la dirección absoluta de un objeto del tipo al cual el puntero está asociado. La descripción de este tipo aparece en el descriptor del puntero. El valor del puntero nil se puede asociar al valor de una dirección falsa, cuya utilización es detectable por el hardware, y podría ser la causa de la activación de un procedimiento especial; así se puede detectar cualquier referencia errónea a través de un puntero con valor nil. Por ejemplo, el valor podría ser una dirección por encima del límite de espacio de memoria disponible.

Los punteros se ubican en la pila de ejecución, de igual forma que otras variables. En Pascal, los objetos referenciados a través de punteros se ubican en la memoria libre. En la Figura 4.14 se observa una visión esquemática de lo anterior para la siguiente declaración.

```
type T = array [0..5] of integer;
var t : T;
```

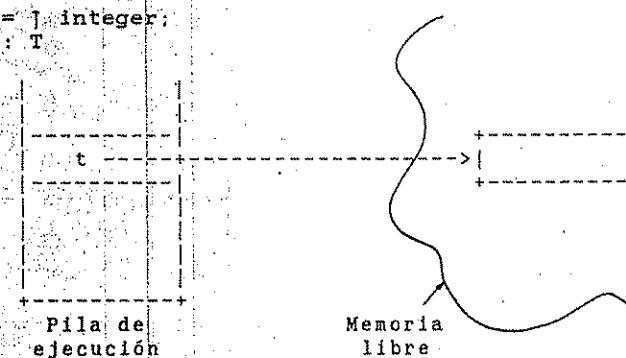


Figura 4.14 Visión en tiempo de ejecución de un puntero de Pascal.

En general, los tipos estructurados se pueden formar agrupando tipos no estructurados en estructuras arbitrariamente

complejas. Como consecuencia, los descriptores de variables de un tipo estructurado se pueden organizar como árboles, describiéndose cada tipo componente por un subárbol. Por ejemplo, un objeto del siguiente tipo t

```
type t = record a: real;
           b: ti;
           c: integer
         end;
```

donde

```
t1 = array [0..3] of integer
```

se puede representar como en la Figura 4.15.

De igual forma, una matriz bidimensional del tipo

```
type t2 = array [0..2] of t4
```

donde

```
t4 = array [0..5] of integer
```

se puede representar como en la Figura 4.16.

Cada componente de tipo t4 de una matriz de tipo t2 se representa por seis valores consecutivos en coma fija. Cada componente elemental de la matriz se puede indexar por un par (i,j), siendo i un valor en el subrango 0..2 que selecciona un componente de tipo t4, y j, un valor en el subrango 0..5 que selecciona un entero dentro de dicho componente. Si b es la primera dirección de la matriz dentro de su registro de activación, la referencia al componente viene dada por la expresión b+6i+j.

4.7.3 "Classes" y Tipos Abstractos de Datos

La implementación de las "classes" de SIMULA 67 se puede hacer de una forma bastante sencilla si no consideramos la posibilidad de instanciación de cuerpos de "class" para ejecutarse de una forma cuasi-concurrente. Las referencias a las "class", como las referencias a las variables en ALGOL 68 o en Pascal, se ubican en la pila de ejecución; las variables que corresponden a los atributos de una "class" se ubican en la memoria libre (ver Ejercicio 4.11).

Otros lenguajes que tienen construcciones derivadas de la "class" de SIMULA 67, tienen un comportamiento diferente en lo que concierne a la generación de objetos encapsulados. A continuación mencionamos brevemente dos enfoques distintos.

		DESCRIPCIÓN	OBJETO
	Tipo	t	+-----> valor en coma flotante
	Constructor	record	+----> valor en coma fija
	Selector	a	valor en coma fija
Campo 1	Tipo	real	valor en coma fija
	Referencia	- - - +	valor en coma fija
	Selector	b	+--> valor en coma fija
Campo 2	Tipo	- - - + +	
	Referencia	- - - + + +	
	Selector	c	
Campo 3	Tipo	entero	
	Referencia	- - - + + + +	
		Descriptor del tipo t1	
		+----+ v	
	Tipo	t1	
	Constructor	array	
	Tipo base	entero	
	Lim. infer.	0	
Índice	Lim. super.	3	
	Tipo de componentes	entero	
	no. de posiciones	1	

Figura 4.15 Un ejemplo de representación de un objeto estructurado jerárquicamente.

En primer lugar está el enfoque más bien estático seguido por Modula y Ada. La idea básica es que el mecanismo de encapsulamiento (module en Modula, package en Ada) debe simplemente empaquetar un conjunto de declaraciones relacionadas entre sí, y proporcionar la inicialización adecuada. El mecanismo de encapsulamiento no presenta un nuevo patrón de un objeto único y estructurado que ha de ser creado dinámicamente, lo que hace es presentar un conjunto de declaraciones visible sólo parcialmente. Entre las entidades exportadas puede haber tipos, pero las variables de estos tipos se comportan como cualquier otras variables del programa.

El enfoque de CLU es más dinámico. Todos los objetos en CLU se ubican en memoria libre y son accesibles en tiempo de ejecución a través de una referencia (que está en la pila). En los lenguajes convencionales, se acostumbra a ver una variable en tiempo de ejecución como un lugar donde se mantiene el valor de un objeto. Sin embargo, en CLU todas las variables son referencias a objetos. Una vez creados, los objetos nunca dejan de existir, aunque pueden llegar a ser inaccesibles si no existen referencias a ellos. La única forma de reutilizar un espacio previamente usado es mediante un recolector de residuos ("garbage collector") que recupera el espacio ocupado por objetos no referenciados por nadie.

Cuanto más dinámico es un lenguaje, más libertad existe para controlar el tiempo de vida y el tamaño de los objetos, pero también más difícil es el manejo de la memoria. Los objetos en memoria libre no se pueden ubicar y desubicar a la entrada y a la salida de un bloque con arreglo a una estrategia "fifo" simple y fácil de implementar, sino que se ubican cuando se crean y se pueden desalojar cuando ya no tienen referencias asociadas.

Dado que la técnica de qué el programador desaloje explícitamente la memoria libre ocupada puede producir el grave problema de las referencias "sueltas", como hemos visto en la Sección 4.5.3 para la primitiva dispose de Pascal, las técnicas eficientes de recolección de residuos se han convertido en un factor clave para la eficiencia de los programas escritos en lenguajes basados en el manejo de la memoria libre. Estas técnicas se discuten en detalle en la sección siguiente.

	DESCRIPCIÓN	OBJETO
Tipo	t2	+--> valor en coma fija
Constructor	array	.
Tipo base	entero	.
del LIM. infer.	0	.
indice		.
LIM. super.	2	"
Tipo de componentes		.
No. de posiciones	6	"
Referencia		
		Descriptor del tipo t4
Tipo	t4	<--
Constructor	array	
Tipo base	entero	
del LIM. infer.	0	
indice		
LIM. super.	5	
Tipo de componentes	entero	
No. de posiciones	1	

Figura 4.16 Un ejemplo de representación de un objeto estructurado jerárquicamente.

4.7.4 Recolección de Residuos

Un recolector de residuos ("garbage collector") es un sistema de ayuda al lenguaje en tiempo de ejecución, que recupera las porciones de memoria libre que hubieran sido ubicadas previamente, y que ya no van a ser utilizadas por el programa. Normalmente el recolector de residuos es invocado automáticamente cuando el espacio disponible para memoria libre está casi acabado.

La recolección de residuos es razonablemente fácil cuando

- i. Los objetos en memoria libre tienen tamaño fijo.
- ii. Se conoce a priori qué campos del objeto contienen punteros a otros objetos de memoria libre.
- iii. Es posible encontrar todos los punteros en la memoria libre.

Los elementos de almacenamiento de tamaño libre se pueden encadenar en una lista de libres; al recolector de residuos se le llama automáticamente cuando existe una solicitud de un nuevo elemento y la lista de libres está vacía.

El siguiente método de recolección en dos pasos se puede implementar fácilmente.

1. Marcar todos los objetos de memoria libre alcanzables, comenzando por las referencias almacenadas en la pila. Para hacerlo se puede utilizar una pila temporal T. Inicialmente T contiene las referencias de la pila a la memoria libre. Cada vez que el elemento más alto de la pila E se extrae de la misma, se marca el objeto referenciado por E, y se reemplaza E por las referencias al (a los) nodo(s) referenciado(s) por E si es que no está(n) marcado(s). Cuando T esté vacía, todos los objetos de memoria libre alcanzables están marcados.
2. Insertar todos los objetos no marcados en la lista de objetos libres.

Este sencillo método presenta en la práctica un cierto número de problemas.

- a. Los objetos de memoria libre son de tamaño variable y contienen punteros a otros objetos que están en diferentes posiciones también de la memoria libre. Por tanto, la identificación de objetos no referenciados puede ser similar al caso de tamaño homogéneo, antes comentado, siempre que la cabecera de cada objeto contenga información codificada (p.ej., un descriptor para tiempo de ejecución) acerca de los campos del objeto que contienen referencias a otros objetos de memoria libre, y acerca del tamaño del objeto.
- b. La pila y la memoria libre a menudo se implementan de forma que crezcan desde los dos extremos de una zona de la memoria de tamaño fijo (Figura 4.17). En este caso, el recolector de residuos entra en juego cuando las dos zonas se encuentran; la zona de memoria libre debe ser recomprimida para permitir que la pila crezca. Sin embargo, la implementación de este método requiere que se disponga de un espacio adicional para ubicar la pila

auxiliar utilizada en el algoritmo de marcaje. Esta pila se puede ubicar en una zona dentro del área de memoria libre, pero su tamaño es muy crítico. Si es grande, se pierde espacio de memoria libre; si es pequeña, hay posibilidad de desbordamiento durante la ejecución del algoritmo de marcaje.

En (Schorr y Waite 1967) se describe un método alternativo que no requiere una pila adicional. A partir de una referencia a la memoria libre almacenada en la pila, se recorre una cadena de objetos de memoria libre hasta el final de la cadena. Cuando se alcanza un elemento de la cadena, se marca, y a su puntero se le hace apuntar al elemento anterior de la cadena. Al llegar al final de la cadena el algoritmo sigue los punteros en sentido inverso. Cuando se encuentra una cadena lateral, se realiza un nuevo camino transversal de una forma similar.

c. Como consecuencia del tamaño variable de los objetos de memoria libre, la solución tipo lista de objetos libres no es muy adecuada pues produciría gran fragmentación de la memoria. Podría ocurrir que no hubiera ningún bloque de posiciones de memoria consecutivas que pudiera albergar un nuevo objeto mientras que la cantidad total de espacio libre fragmentado es mayor que la cantidad solicitada. Se hace necesaria una compactación del espacio libre. La principal dificultad está en que los punteros a los objetos activos deben ser modificados cuando dichos objetos se trasladan a posiciones diferentes.

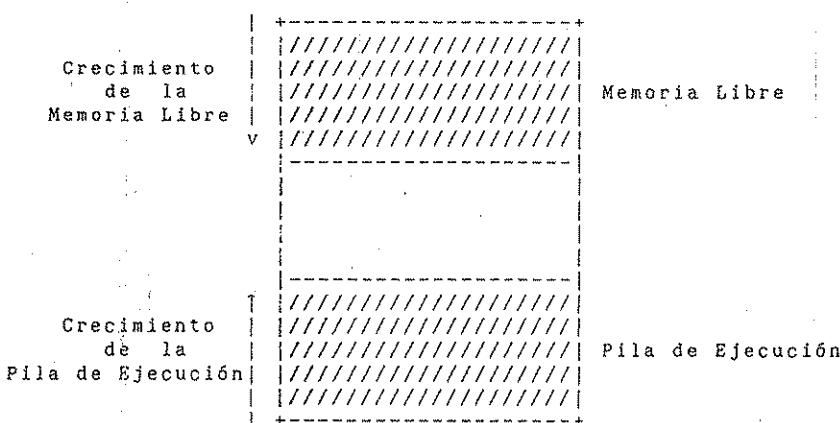


Figura 4.17 Organización de la memoria principal.

El principal problema con la recolección de residuos es que se pierde mucho del "apreciado" tiempo de proceso cada vez que

entra en juego el recolector. Esto puede ser especialmente grave en los sistemas de tiempo real, ya que puede llegar una petición de un servicio urgente por parte del entorno justo después de que el recolector de residuos haya comenzado su complejo trabajo.

La recolección de residuos se puede distribuir más uniformemente a través del tiempo de proceso utilizando un esquema de conteo de referencias. En un esquema de este tipo, cada objeto lleva en un contador la cuenta del número de objetos que le referencian. Cuando ese contador llega a cero, se puede liberar el área destinada al objeto. Desgraciadamente, este método no funciona en el caso de que los objetos se encadenen en una cola circular. Cuando no hay referencias desde la pila a la estructura circular, los contadores de referencias asociados a cada nodo de la lista no valen cero, sino uno. En la sección de Sugerencias para Ampliación Bibliográfica del final de este capítulo, se citan algunas soluciones a este problema.

Finalmente, debemos mencionar que la recolección de residuos también se puede ver como un proceso que está activo concurrentemente con el programa en ejecución. Las técnicas de recolección en paralelo podrían llegar a ser de interés práctico con el advenimiento de las arquitecturas multiprocesador. Un multiprocesador dedicado a la ejecución de un lenguaje de alto nivel con filosofía de memoria libre, podría reservar uno de los procesadores para realizar "on-line" la recolección de los residuos de memoria. La tendencia a la baja del coste de los procesadores hará probablemente esta solución más atractiva en el futuro de lo que lo es hoy en día.

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

La visión sistemática de los agregados de datos y la clasificación de los métodos de estructuración presentadas en la Sección 4.2 están tomadas de C.A.R. Hoare en (Dahl y otros 1972). Hoare define los productos cartesianos, las aplicaciones finitas, las secuencias, las uniones discriminadas, las estructuras de datos recursivas y los conjuntos potencia. También discute los modelos de implementación de cada mecanismo de estructuración.

(Tanenbaum 1978) y D.M. Berry en (Wegner 1979) presentan una comparación crítica de ALGOL 68 y Pascal, con especial atención a sus estructuras de tipos. (Habermann 1973), (Lecarme y Desjardins 1975), (Welsh y otros 1977) y (Tennent 1978) contienen varios comentarios sobre Pascal y muestran en particular las inseguridades de su estructura de tipos. (Fischer y LeBlanc 1980) discuten aspectos de implementación de la comprobación de tipos en los programas escritos en Pascal.

Los tipos abstractos de datos fueron presentados en (Liskov y Zilles 1974). En (Liskov y Zilles 1975), (Wulf y otros 1976), (Guttag 1977), (Goguen y otros 1978) y (Guttag y otros 1978) se presentan aspectos formales de la especificación de los tipos

abstractos de datos. (Gries y Gehani 1977) contiene una introducción al concepto de tipo de dato genérico. La publicación de la conferencia de (ACM-SIGPLAN 1976) contiene muchos artículos sobre tipos de datos y abstracción de datos. El artículo de P. Wegner (Wegner 1979) presenta una visión general de conceptos relacionados con los tipos de datos en los lenguajes de programación.

Además de CLU y Ada, hay otros lenguajes que proporcionan facilidades para la abstracción de datos, por ejemplo, Euclid, Gypsy, Modula, Mesa, Alphard y Russell. Alphard (Shaw y otros 1977), (Wulf y otros 1976) y Russell (Demers y Donahue 1980a), (Demers y Donahue 1980b) son particularmente interesantes.

Los modelos de implementación de los objetos se analizan en el artículo antes mencionado de C.A.R. Hoare en (Dahl y otros 1972), así como en la mayoría de los libros de texto para el diseño de compiladores, como (Gries 1971), (Aho y Ullman 1977) y (Barrett y Couch 1979), y en otros libros de texto de lenguajes de programación, como (Pratt 1975). La recolección de residuos se estudia en (Schorr y Waite 1987), (Knuth 1973) y (Deutsch y Bobrow 1976). (Pratt 1975) contiene un repaso de las técnicas de recolección de residuos para los lenguajes basados en almacenamiento dinámico. El artículo de U. Hill en (Bauer y Eickel 1976) presenta una discusión detallada de ciertas técnicas especiales en tiempo de ejecución para ALGOL 68. En (Steele 1975) y (Dijkstra y otros 1978) se discute la recolección de residuos paralela.

Ejercicios

- 4.1 ¿Cuáles son las diferencias entre las facilidades para la definición de tipos ofrecidas por Pascal (y ALGOL 68) y los tipos abstractos de datos?
- 4.2 ¿Qué significa que un lenguaje haga una comprobación rigurosa de tipos ("strong typing"), y cuáles son los beneficios que se obtienen de ello?
- 4.3 Dar algunos ejemplos que demuestren que Pascal no es un lenguaje que realice una comprobación rigurosa de tipos.
- 4.4 Diseñar una serie de programas sencillos que comprueben las reglas de compatibilidad de tipos adoptadas por el compilador de Pascal que Vd. utilice.
- 4.5 Diseñar una serie de programas sencillos que verifiquen la seguridad de los registros variantes en el compilador de Pascal que Vd. utilice normalmente.
- 4.6 Diseñar una serie de programas sencillos que evalúen la seguridad en el manejo de punteros en el compilador de ALGOL 68 que Vd. utilice habitualmente.

- 4.7 Diseñar una serie de programas sencillos que evalúen la seguridad en el manejo de punteros en el compilador de Pascal que Vd. utilice habitualmente.
- 4.8 Discutir los aspectos problemáticos de los tipos por enumeración en Pascal siguiendo las críticas aportadas en (Welsh y otros 1977) y (Tennent 1978). Preparar programas sencillos que clarifiquen las opiniones.
- 4.9 El modo:
`union (int,ref,int)`
 ?Es legal en ALGOL 68? ?Por qué? ?Hay un problema similar en Pascal?
- 4.10 La "class" pila declarada en la Sección 4.6.1 tiene un atributo `primero`, que se utiliza para apuntar al primer elemento de la pila, y los cuatro atributos `sacar`, `meter`, `cima` y `vacia`, que se utilizan para operar sobre la copia de la pila. Ya discutimos el hecho de que `primero` es accesible directamente desde fuera de la "class", y por tanto, ésta no está protegida. Todas las variables y los procedimientos declarados en el bloque exterior de una "class" son accesibles desde fuera de la misma. ?Por qué no es posible hacer a `primero` inaccesible declarándolo como una variable local a un bloque interno de la "class"? ?Se puede hacer local a uno de los procedimientos?
- 4.11 Las "class" se pueden implementar asociando un registro de activación a cada una de ellas. Para cada atributo, el registro de activación contiene la suficiente cantidad de memoria para almacenar su valor si es una variable, y un puntero al segmento de código si es un procedimiento. El segmento de código correspondiente a la "class" implementa el código de inicialización. La ejecución de una sentencia `new` origina la ubicación de un registro de activación para la "class" y la ejecución del código de inicialización.
 - a. ?Se pueden ubicar en una pila los registros de activación de las "class"?
 - b. ?Cada copia de una "class" requiere su propia copia de los procedimientos que son atributos de la "class"?
- 4.12 ?Cómo se pueden implementar las uniones discriminadas en SIMULA 67? ?Cómo se puede utilizar la sentencia `INSPECT` de SIMULA 67 para trabajar con las uniones discriminadas de una manera segura? ?Cuáles son las diferencias entre esta sentencia y la cláusula de conformidad de ALGOL 68?
- 4.13 ?Cuáles son las diferencias entre las "class" de SIMULA 67 y los tipos abstractos de datos?

4.14 ?Cuáles son las diferencias semánticas básicas entre el "cluster" de CLU y el "package" de Ada.

4.15 Se dan dos "cluster" de CLU con las siguientes cabeceras:

PILA = cluster is CREAR, METER, SACAR, CIMA, VACIA
 COLA = cluster is CREAR, ENCOLAR, DESENCOLAR, PRIMERO, VACIA

El "cluster" PILA define estructuras tipo LIFO. El "cluster" COLA define estructuras tipo FIFO. Los dos tipos abstractos de datos pueden almacenar elementos del mismo tipo (p.ej., enteros).

a. Diseñar una función IGUAL que dé un resultado cierto si la secuencia de elementos extraídos de una pila (a través de la operación CIMA) y los extraídos de una cola (a través de la operación PRIMERO) son los mismos.

b. La función IGUAL sería más eficiente si tuviera acceso directo a las estructuras de datos (vectores) utilizadas para representar pilas y colas. ?Cómo podría resolver este problema una implementación de Ada?

4.16 Sea X una matriz n-dimensional de Pascal declarada en el procedimiento P, y sea d su desplazamiento dentro del registro de activación de P. Sean i11,i12,...,lin y ls1,ls2,...,lsn respectivamente los límites inferior y superior de los n subíndices. ?Cómo se pueden evaluar en tiempo de ejecución las direcciones de X[i1,i2,...,in]? (Nota: Supóngase que X está almacenada de tal forma que es el índice primero el que crece más lentamente).

4.17 ?Cómo se puede utilizar el modo bits de ALGOL 68 para implementar los conjuntos potencia?

4.18 Supongamos que X es una "subclass" de Y, es decir, X ha sido declarado como Y class X begin...end. Dadas las dos declaraciones

```
ref (X) a;
ref (Y) b;
```

SIMULA 67 permite las dos siguientes sentencias de asignación

```
a:-b;
b:-a;
```

Una de estas dos sentencias se puede utilizar para demostrar que SIMULA 67 no es un lenguaje con rigurosa comprobación de tipos. ?Cuál es? (Nota: considerar los atributos de acceso de la "class" después de cada asignación). Proponer una regla que prevenga contra tales asignaciones. ?Hay alguna

razón para no utilizar la regla?

4.19 ALGOL 68 y Euclid adoptan equivalencia de tipos por estructura. Como ya se mencionó en la Sección 4.5.2, los tipos definidos recursivamente podrían producir que el algoritmo de comprobación de tipos entrara en un bucle sin fin. ?Cómo podrían evitar el problema estos lenguajes?

**ESTRUCTURAS
DE
CONTROL.**

Este capítulo hace un análisis detallado de las estructuras de control, es decir, de los mecanismos por medio de los cuales los programadores pueden especificar el flujo de ejecución entre los componentes de un programa. En la Sección 2.3 distinguimos entre dos tipos de estructuras de control: a nivel de unidad y a nivel de sentencia. Por su mayor sencillez, empezaremos el análisis por el segundo tipo. Las estructuras de control a nivel de unidad y su implementación se estudiarán en la Sección 5.2.

5.1 ESTRUCTURAS DE CONTROL A NIVEL DE SENTENCIA

Hay tres clases de estructuras de control a nivel de sentencia: secuencia, selección e iteración. Analizaremos cada una de ellas por separado en las secciones 5.1.1 a 5.1.3.

Las estructuras de control a nivel de sentencia contribuyen en gran medida a la legibilidad y mantenimiento de los programas. De hecho, es esencial para un uso intelectual de la programación que las instrucciones estén ligadas de acuerdo con esquemas suficientemente sencillos, naturales y fáciles de entender.

5.1.1 Secuencia

La secuencia es el mecanismo de estructuración más simple del que disponen los lenguajes de programación. Se usa para indicar que una sentencia (B) se ejecutará a continuación de otra sentencia (A). La notación usual es

A;B

donde ";" (que puede leerse como "y después") indica el operador de control de secuencia. Los lenguajes orientados a líneas (p.ej., FORTRAN) utilizan implícitamente el fin de línea para separar instrucciones e imponer un mecanismo de secuencia entre ellas. Es posible agrupar un conjunto de sentencias en una secuencia para formar una única sentencia compuesta. Algunos lenguajes (p.ej., ALGOL 60 y Pascal) utilizan las palabras reservadas begin y end para este fin. Por ejemplo:

begin A;B;...end.

5.1.2 Selección

La estructura de control selección permite al programador especificar que ha hecho una elección entre un cierto número de posibles sentencias alternativas.

La sentencia IF lógica de FORTRAN es un ejemplo de sentencia de selección que especifica la ejecución de una sentencia según una expresión de tipo lógico. Por ejemplo, la sentencia FORTRAN

IF (I.GT.0) I = I-1

decrementa el valor de *I* en 1 si *I* es positivo.

Más amplia y potente es la sentencia if de los lenguajes tipo ALGOL, en la que la presencia de una rama else permite al programador escoger entre dos alternativas como consecuencia de una pregunta. Aquí, a diferencia del FORTRAN, una rama puede ser cualquier sentencia (p.ej., una sentencia compuesta). Por ejemplo, el fragmento de programa

```
if i = 0
  then i := j
  else begin i := i+1;
           j := j-1
      end
```

pone *i* al valor de *j* si *i*=0; de lo contrario, incrementa *i* en 1 y decrementa *j* en 1. Si se hubieran omitido las palabras reservadas begin y end, *j := j-1* se consideraría como una sentencia siguiente a la selección, ejecutándose también para el caso de *i*=0.

La construcción de selección en ALGOL 60 presenta un conocido problema de ambigüedad. En el ejemplo

```
if x>0 then if x<10 then x := 0 else x := 1000 end
```

no está claro si la rama else corresponde a la condición más interna (if *x*<10...) o a la más externa (if *x*>0...). Según una interpretación, la ejecución de la sentencia para *x*=15 asignará 1000 a *x*; y según la otra, *x* permanecerá invariable. Para eliminar la ambigüedad, la sintaxis de ALGOL 60 requiere una sentencia incindicional en la rama then de una sentencia if. De este modo, la sentencia anterior quedaría

(i) if *x*>0 then begin if *x*<10 then *x* := 0 else *x*:=1000 end

o también

(ii) if *x*>0 then begin if *x*<10 then *x* := 0 end else *x* := 1000

según la interpretación que se deseé.

El mismo problema se resuelve en PL/I y Pascal casando de forma automática cada rama else con la condición más próxima que no tenga else. De esta forma, los delimitadores begin y end sobran en el caso (i). Esta regla de concurrención se indica explícitamente en la definición del lenguaje. Aunque si bien la regla elimina la ambigüedad, las estructuras con muchas condiciones anidadas son difíciles de leer, especialmente si el programa se ha escrito sin un decalaje esmerado. En este caso, será conveniente el uso de los delimitadores begin y end para hacer más explícita la interpretación deseada.

En ALGOL 68 se adopta una variación sintáctica que evita este problema usando la palabra reservada fi como un delimitador

de cierre de la sentencia if. Se permite una secuencia de sentencias entre las palabras reservadas then y else y entre else y fi. Con esto se hace innecesario el uso de los delimitadores begin y end para construir una sentencia compuesta. De este modo, los ejemplos anteriores se pueden codificar en ALGOL 68 como

```
if i = 0
  then i := j
  else i := i+1;
      j := j-1
fi
y
if x>0 then if x<10 then x := 0 else x := 1000 fi fi
a
if x>0 then if x<10 then x := 0 fi else x := 1000 fi
```

según la interpretación deseada. Ada adopta una solución similar mediante el uso de la palabra reservada end if para cerrar una selección.

Tanto ALGOL 68 como ADA permiten al programador el uso de abreviaturas en caso de poder elegir entre muchas alternativas, dependiendo de condiciones diferentes. Por ejemplo, el engorroso fragmento

```
if a
  then S1
  else if b
    then S2
    else if c
      then S3
      else S4
    fi
fi
```

puede escribirse en ALGOL 68 usando la contracción elif (Ada utiliza elseif) y eliminando el fi (end if en Ada) de las cláusulas internas.

```
if a
  then S1
elif b
  then S2
elif c
  then S3
  else S4
fi
```

Recientemente, PL/I ha adoptado la construcción especial

select con el fin de especificar la selección entre dos o más opciones. En PL/I, el ejemplo anterior quedaría:

```
SELECT
  WHEN (A) S1;
  WHEN (B) S2;
  WHEN (C) S3;

  OTHERWISE S4;
END;
```

Las construcciones de selección con múltiple alternativa se expresan en otros lenguajes como ALGOL 68, Pascal y Ada, con la sentencia case, la cual especifica la selección de una rama según el valor de una expresión. Por ejemplo: el siguiente fragmento de Pascal evalúa resultado a partir de oper1 y de oper2, según el operador lógico especificado por el valor de la variable operador:

```
var operador: char;
oper1,oper2,resultado: boolean;

case operador of
  ".": resultado := oper1 and oper2;
  "+": resultado := oper1 or oper2;
  "=": resultado := oper1=oper2;
end
```

La selección de una rama de un case en ALGOL 68 sólo puede basarse en el valor de una expresión de tipo entero. El valor de cada expresión, -i-, selecciona la rama i-ésima para ejecutarla. Por otro lado, la selección en Pascal puede basarse en el valor de una expresión de cualquier tipo ordinal, estando etiquetadas explícitamente las ramas con uno o más de los valores que puede tomar la expresión. Por consiguiente, el orden en que aparezcan las ramas es indiferente.

La mayor potencia expresiva del case de Pascal tiene por contra la posibilidad de producir programas inseguros. Pascal no especifica el efecto de un valor de la expresión de selección que está fuera del rango de los valores indicados explícitamente, y no permite tampoco especificar en la construcción qué acción se habría de tomar en ese caso. En ALGOL 68 se puede especificar la cláusula opcional out, que se ejecuta cuando el valor de la expresión de selección no está expresado en el conjunto de valores. Si se omite la cláusula out, equivale a una cláusula out con una sentencia skip, lo cual es equivalente a una sentencia nula.

La selección múltiple de Ada combina los aspectos positivos del Pascal y del ALGOL 68. En primer lugar, la selección de rama puede hacerse con expresiones de tipo entero o enumeración. Además es preciso que se estipulen en las selecciones todos los

valores posibles del tipo de la expresión. Por último, se puede utilizar la abreviatura others para representar aquellos valores que no se han indicado explícitamente. Por tanto, la selección múltiple del ejemplo anterior quedaría codificada en Ada como sigue:

```
case OPERADOR of
  when ".":> RESULTADO := OPER1 and OPER2;
  when "+":> RESULTADO := OPER1 or OPER2;
  when "=":> RESULTADO := OPER1=OPER2;
  when others:> ... mensaje de error...
end case;
```

Dijkstra (Dijkstra 1976) propuso un mecanismo sencillo y potente para especificar una selección. El formato general de la construcción es

```
if B1 --> S1
  B2 --> S2
  ...
  Bn --> Sn
fi
```

donde Bi, tal que $1 \leq i \leq n$, es una expresión lógica denominada guarda, y Si, siendo $1 \leq i \leq n$, es una lista de sentencias. A Bi --> Si, para $1 \leq i \leq n$, se le denomina comando de guarda. La semántica de la sentencia if es que puede elegirse aleatoriamente para ser ejecutada cualquier Si cuyo Bi sea cierto. Si ningún guarda es cierto, se aborta el programa; por lo tanto, el programador se ve forzado a indicar todas las alternativas posibles.

La característica más novedosa e interesante de la construcción de Dijkstra es la abstracción que puede proporcionar la aleatoriedad. El programador no está obligado a especificar detalladamente programas en los que es indiferente la alternativa que se escoja en un momento dado. Por ejemplo, el mayor de dos números A y B se puede calcular de la siguiente forma:

```
if A>=B --> MAX := A
  A<=B --> MAX := B
fi
```

En la Sección 5.2.4.3 veremos que en Ada hay un formato especial de sentencia if con guarda.

5.1.3 Iteración

Los cálculos más usuales implican la repetición de un cierto número de acciones. Por consiguiente, todos los lenguajes de alto nivel proporcionan estructuras de control que permiten al programador especificar bucles sobre todo tipo de instrucciones.

FORTRAN proporciona la sentencia DO, con la cual se puede indicar un número fijo de iteraciones mediante la introducción de un contador (variable de control del bucle) que toma valores entre un conjunto finito de enteros. El siguiente ejemplo:

```
DO 7 I = 1,10
A(I) = 0
B(I) = 0
7 CONTINUE
```

pone a cero los elementos con subíndice 1 a 10 de las matrices A y B. Las estructuras de control iterativas mediante contadores son muy útiles y han sido adoptadas por muchos lenguajes, incluidos COBOL, ALGOL 60 y PL/I. Pascal permite bucles donde la variable de control puede ser de cualquier tipo ordinal, como puede verse en el ejemplo siguiente:

```
type dia=(lun,mar,mie,jue,vie,sab,dom);
var dia_semana:dia;
```

```
for dia_semana := lun to vie do
```

y para recorrer los días en orden inverso,

```
for dia_semana := vie downto lun do...
```

Pascal no permite el que se altere en el bucle ni la variable de control ni sus límites superior e inferior. Esta restricción, lejos de imponer restricciones al programador, contribuye a la legibilidad y mantenimiento de los programas, limitando el alcance del contador del bucle.

Se puede utilizar como modelo de repetición sobre un conjunto finito de valores el caso frecuente en el cual conocemos de antemano el número de iteraciones, como por ejemplo, en el proceso de matrices. Sin embargo, a menudo no conocemos con antelación el número de repeticiones. Por ejemplo, un programa que procesa todos los registros de un fichero normalmente no conoce el número de elementos del fichero. Por esta razón, muchos lenguajes posteriores al FORTRAN proporcionan estructuras de iteración según condición, es decir, estructuras en las que se ejecutan iteraciones hasta que cambia el valor de una expresión lógica.

Pascal soporta dos estructuras de este tipo: la primera (while), describe cualquier número de iteraciones incluido el cero. La segunda (until), describe bucles con al menos una iteración. El programa citado anteriormente para procesar un fichero, quedaría en Pascal:

```
while not eof(fich) do
begin
  "leer registro del fichero fich";
  "procesar registro leido"
end
```

La condición de fin_de_fichero eof se evalúa antes de ejecutar el cuerpo del bucle, saliéndose del mismo, si not eof(fich) es falso. Por tanto, el programa también funcionará en el caso de que el fichero esté vacío.

La estructura until es similar, excepto en que la condición de salida se comprueba al final del cuerpo del bucle. En el caso de que fich tenga siempre un elemento al menos, el ejemplo anterior se podría escribir

```
repeat
  "leer registro del fichero fich";
  "procesar registro leido"
until eof(fich);
```

En este fragmento, el bucle se repite mientras la condición eof(fich) es falsa.

Las estructuras de control mediante contador o según condición de Pascal, las combinan PL/I y ALGOL 68 en una forma única compuesta de varias partes opcionales. El formato general del bucle en ALGOL 68 es

```
for i from j by k to m while b do . . . od
```

donde do y od encierran el cuerpo del bucle y for, from, by, to y while son cláusulasopcionales. Si se omite la cláusula for tenemos una variable de control implícita. Si se omite from asume el valor 1 como inicial. Por último, si se omite la cláusula by, el valor a incrementar después de cada iteración es 1. Por ejemplo, la sentencia

```
to 10 do B:= B+2 od
```

```
suma 20 a B.
```

Quitando las opciones for, from, by y to tenemos un bucle según condición; quitando la opción while tenemos un bucle sin fin del que se puede salir explícitamente desde dentro del bucle.

En ALGOL 68 (como en Pascal), la variable de control del bucle no se puede modificar dentro del mismo, y su alcance se limita al cuerpo del bucle, siendo por tanto inaccesible fuera del mismo. Por el contrario, los valores de las expresiones que siguen a by y to pueden ser alteradas dentro del bucle. Sin embargo, esas expresiones sólo se evalúan una vez, antes de que comience la ejecución del bucle; por tanto, cualquier cambio de dichas expresiones no afecta a la condición de terminación del

mismo.

Ada posee una sola estructura de iteración con el siguiente formato:

```
especificacion de iteracion loop
    cuerpo del bucle
end loop
```

donde la especificación de iteración puede ser
while condicion

```
o
for contador in rango_discreto
o
for contador in reverse rango_discreto
```

Además, el bucle puede acabarse con una sentencia de salida incondicional

```
exit;
o por una sentencia condicional
exit when condicion;
```

Si el bucle está anidado dentro de otros bucles, es posible salir del bucle más interno de la forma siguiente:

```
BUCLE_PRINCIPAL:
    loop
        .
        .
        loop
            exit BUCLE_PRINCIPAL when A=0;
        end loop;
        .
    end loop BUCLE_PRINCIPAL;
```

-- aquí continúa la ejecución después de la sentencia exit.

En el ejemplo anterior, cuando A llega a valor cero en el bucle más interno, el control se transfiere a la sentencia siguiente a end loop BUCLE_PRINCIPAL.

La sentencia exit se utiliza para especificar la terminación prematura de un bucle. Tiene el mismo efecto que la sentencia LEAVE de PL/I.

PLZ proporciona un ejemplo interesante de estructuras iterativas. El cuerpo del bucle está acotado por los delimitadores do, od. La sentencia exit permite especificar una salida desde cualquiera de los bucles anidados. Se puede utilizar una sentencia repeat para terminar la iteración en curso de cualquiera de los bucles internos y reiniciar su ejecución desde la primera instrucción del cuerpo del bucle. Una sentencia repeat o exit que no especifique la etiqueta de un bucle hará referencia por defecto al bucle interno más inmediato. Por ejemplo, veamos un programa que calcula el número (n) de filas (de longitud m) de una matriz a de dimensiones k x m que son iguales, elemento a elemento, a una matriz b unidimensional de longitud m.

```
i := 0
n := 0
principal: do
    i := i+1
    if i>k then exit fi
    j := 0
    do
        j := j+1
        if j>m then exit fi
        if a[i,j] = b[j] then repeat from principal fi
    od
    n := n+1
od
```

El lenguaje C posee una construcción similar.

Utilizando la notación de comandos con guarda de Dijkstra, los bucles se especifican acotando los comandos entre las palabras reservadas do, od.

```
do B1 --> S1
  B2 --> S2
  .
  .
  Bn --> Sn
od
```

La semántica de la sentencia es tal que en cada iteración se ejecuta una sentencia Si, aquella cuyo guarda Bi sea cierto. Si uno o más guardas son ciertos, la elección será aleatoria. Si ninguno lo es, termina el bucle. Las sentencias de selección e iteración basadas en comandos con guarda estimulan la producción de algoritmos elegantes y correctamente estructurados. Además, tal y como veremos en la Sección 6.4, los programas escritos con la citada notación facilitan los razonamientos formales en la corrección de programas. Sin embargo, esta notación no ha sido adoptada por ninguno de los lenguajes existentes.

5.1.4. Medida de las Estructuras de Control a Nivel de Sentencia

La secuencia, selección e iteración son para el programador las herramientas de estructuración básicas para controlar y organizar el flujo entre sentencias. La secuencia es una abstracción sobre la búsqueda secuencial de instrucciones que proporciona el contador de programa del ordenador. La selección y la iteración son abstracciones sobre el mecanismo de control, sencillo pero excesivamente poderoso, que proporciona el hardware para modificar explícitamente el valor del contador de programa; es decir, saltos condicionales e incondicionales.

Hay diversos motivos por los que se prefieren estructuras de control abstractas frente a mecanismos de bajo nivel de transferencia explícita de control. Primero, están más orientados al problema y de esta forma los programadores pueden expresar sus intenciones con mayor facilidad utilizando modelos generales de secuencia, selección e iteración. Segundo, si se usan saltos explícitos, los programas adoptarán fácilmente la bien conocida estructura de madeja, según se ve en la Figura 5.1. Evidentemente, las estructuras de alto nivel se traducen a saltos condicionales e incondicionales en código máquina, pero este hecho no concierne al programador. El peso de producir código máquina optimizado se deja por completo al traductor.

Actualmente hay un consenso general de que, incluso en aplicaciones cuyo objetivo primario es la eficiencia, la poca claridad que engendra el uso de estructuras de control a nivel de máquina es un precio demasiado alto. Ha habido propuestas para añadir estructuras de alto nivel a los lenguajes de nivel de máquina existentes; últimamente, se han trasladado a los lenguajes de alto nivel para aplicaciones y sistemas. (p.ej., Bliss, C, PLZ, Euclid, Ada).

Sin embargo, tal y como vimos en la Secciones 5.1.2 y 5.1.3, se han propuesto una gran variedad de nuevas estructuras de control y aún no hay acuerdo sobre cuál es la mejor para incluirla en un lenguaje. No obstante, aunque la secuencia, selección (*if then else*) e iteración (*do while*) proporcionan en teoría un conjunto de estructuras que permiten codificar cualquier algoritmo (Böhm y Jacopini 1966), el uso exclusivo de esas tres estructuras es a menudo artificioso, obteniéndose como resultado un programa desgarbado. Otras estructuras adicionales, redundantes en teoría, como la selección múltiple (*case*), bucles según contador (*for*) y bucles según condición (*until*) pueden facilitar la escritura y legibilidad de los programas.



Figura 5.1 Estructura de spaghetti.

Un caso típico de la no idoneidad práctica del conjunto restringido de estructuras de control surge cuando es necesario salir del interior de un bucle. Como ejemplo, supóngase un programa que procesa un paquete de tarjetas, cada una de las cuales contiene n valores enteros. La suma de los n valores es cero para la última tarjeta. Si la suma no es cero, los n valores se toman como válidos y se procesan. De una manera abstracta, la solución más simple sería la siguiente:

```
do
    leer N valores y sumarlos;
    si la suma es cero, acabar el bucle;
    procesar los N valores
od
```

Utilizando la estructura *while*, se llega a una duplicidad de sentencias, como vemos a continuación:

```
leer N valores y sumarlos;
while suma <> 0 do
    begin
        procesar los N valores;
        leer N valores y sumarlos
    end
```

De una forma similar, utilizando la estructura *repeat...until*, lo que precisamos duplicar es la evaluación de la condición:

```
repeat
    leer N valores y sumarlos;
    if suma <> 0 then procesar los N valores
until SUMA = 0
```

La solución inicial, la cual usa una transferencia de control explícita para salir del bucle a la mitad, es más natural, más fácil de escribir y entender, y su implementación es más eficiente. Los bucles con salida en el centro de la ejecución son fácilmente codificables con la sentencia *goto*. Puesto que el uso a discreción de la sentencia *goto* puede conducir a programas mal estructurados, pero es preciso un uso restrictivo en casos como la salida de un bucle, diversos lenguajes (como Ada)

proporcionan una construcción específica (*exit*) para estos casos. Dicha sentencia *exit* proporciona una forma restrictiva (y segura) de *goto* hacia adelante, que sólo puede utilizarse para terminar un bucle. En el Ejercicio 5.4 se da un formato más general de la sentencia *exit*, pero aún más restrictivo que la sentencia *goto*.

Hay también ocasiones en las que se desea terminar la iteración en curso en el caso de darse alguna condición durante la ejecución del cuerpo del bucle. Esta situación se puede codificar en PLZ (o en C) con la sentencia *repeat* (en C, *continue*). Considérese el ejemplo siguiente, en el cual cuando *x=true* deben terminar los bucles B y A, y comenzar una nueva iteración A:

```
A: do...
  B: do...
    if x then repeat from A
    ...
    od
  ...
od
```

El mismo ejemplo, pero en Ada, queda

```
A: loop...
  B: do...
    exit when x;
  end loop;
  if not x then
    ...
  end if
end loop
```

La solución de Ada implica una reevaluación innecesaria de la condición de salida nada más salir del bucle interno; para evitar esto, se puede optar por utilizar la sentencia *goto*.

Este fenómeno está generalizado. La carencia de estructuras de control especializadas para la resolución de un problema puede llevar a soluciones artificiosas y poco eficientes si no disponemos de la sentencia *goto*. Por otro lado, no sabemos con exactitud qué estructuras de control serían necesarias para cubrir todas las posibilidades de la programación. Muchos lenguajes proporcionan un amplio abanico de estructuras disciplinadas. A veces, también disponen de una sentencia *goto* sin restricciones. La controversia está en dilucidar si están o no bien diseñados. Si los programas están diseñados según una disciplina, la utilización de la sentencia *goto* surge sólo ocasionalmente. Sin embargo, cuando se dispone de la sentencia *goto*, no hay razón para restringir su uso, con lo que los programadores corren el riesgo de producir programas embrollados.

Empero, la solución parcial consiste en adoptar una programación estándar que se auxilie de *goto* sólo para aquellos casos "legítimos" en los que el lenguaje no proporciona las estructuras necesarias. La legitimidad puede definirse en base a las necesidades de la aplicación.

En resumen, influyen muchos factores en la elección del conjunto de estructuras de control a incluir en un lenguaje, siendo difícil discutir en términos absolutos los méritos de las diferentes soluciones. Por ejemplo, el mínimo conjunto de estructuras de control que parece ser un objetivo razonable, en tanto en cuanto concierne a la facilidad de aprender un lenguaje, choca con otros objetivos como puede ser la fuerza expresiva.

5.1.5 Estructuras de Control Definidas por el Usuario

Una propiedad interesante del bucle según contador de Pascal es que la variable de control puede ser de cualquier tipo ordinal, no limitada a un subconjunto de enteros. Lenguajes como CLU, Alphard y Euclid poseen abstracciones adicionales que permiten al programador definir variables de control de cualquier tipo abstracto de datos, y proporcionan construcciones para especificar cómo va a ser la secuencia de valores de cada variable de control. Los lenguajes citados se pueden considerar como extensibles, ya que el usuario puede aumentar la base del lenguaje mediante la definición de nuevos tipos de datos (abstractos), nuevas operaciones (con procedimientos) e, incluso, nuevas estructuras de control. Nuestro estudio de las estructuras de control definidas por el usuario se basará en las construcciones que proporciona CLU.

En el fragmento de programa CLU

```
for elemento: nodo in lista(x) do
  ejecutar una acción sobre elemento
```

elemento, la variable de control, es de tipo *nodo* definido por el usuario y lista es un iterador, que es una unidad de programa particular que produce la secuencia de valores a utilizar en el bucle *for*. El fragmento anterior lo que hace es coger y manipular todos los nodos pertenecientes a la lista x.

El iterador suministra cada vez un elemento del conjunto (en el ejemplo, una lista). La política de selección del siguiente elemento del conjunto es desconocida para el usuario del iterador y está soportada por el propio iterador. El iterador del ejemplo se puede especificar mediante un módulo que toma un parámetro de tipo lista encadenada y devuelve un parámetro de tipo *nodo*:

```

lista = iter (z:lista_enlazada) yields (nodo)
:
:
yield (n)
:
:
end lista

```

En cada iteración del bucle (`for elemento...`) se activa el módulo `lista`, dando un nodo que con la operación `yield` se asigna a `elemento`. (recordar que CLU tiene asignación por referencia (Sección 4.6.2.1) y que, por tanto, `elemento` indica al mismo objeto devuelto por `lista`). Como consecuencia de la cesión se define el iterador, se guarda su entorno local, y el control de la ejecución regresa a la llamada del bucle. En cada repetición del bucle se reanuda el iterador en su entorno local, como una corutina. La sentencia `for` termina cuando el iterador indica que la secuencia de objetos se ha agotado.

Los iteradores de CLU sólo son llamados por sentencias `for`, y cada `for` llama a un único iterador. De este modo, siempre se anidan las activations de las llamadas, pudiéndose soportar fácilmente mediante una pila, según vimos anteriormente.

5.1.5.1 Implementación de Modelos para Iteradores

Se pueden diferenciar cuatro acciones asociadas a un iterador:

- * Llamada al iterador, que corresponde a la primera activation del iterador provocada por el comienzo de un bucle `for`.
- * Reanudación del iterador, que corresponde a cualquier activation del iterador después de la primera iteración.
- * Cesión, es decir, la acción de devolver el control al programa que ejecuta la sentencia `for`.
- * Regreso desde el iterador, cuando se alcanza el `end` del mismo.

El registro de activation del iterador contiene dos puntos de retorno y un enlace dinámico. El primer punto de retorno (punto de retorno normal) es la dirección de la primera instrucción del cuerpo del bucle `for` que ha llamado al iterador. El segundo punto de retorno (punto de retorno final) es la dirección de la instrucción siguiente al bucle `for`, es decir, la instrucción que se ejecutará cuando acabe el iterador. El enlace dinámico apunta al registro de activation de la unidad que ejecuta el bucle. El registro de activation de dicha unidad

contiene una entrada adicional (enlace de reanudación) que se utiliza para reunir conjuntamente en una cadena de iteradores la información necesaria para reanudar los iteradores interrumpidos que fueron llamados por cualquier módulo.

La llamada a un iterador se asemeja a la llamada a un subprograma convencional: se pone en la pila el registro de activation del iterador y se guardan los puntos de retorno y el enlace dinámico.

La cesión difiere del retorno desde un procedimiento normal, dado que el registro de activation del iterador permanece en la pila. En primer lugar, se pone en la pila la base de reanudación. Dicha base contiene la información necesaria para reanudar el iterador en la siguiente iteración, es decir, la dirección de la instrucción (punto de retorno) a donde vuelve el control una vez reanudado el iterador, y una referencia al registro de activation del iterador (enlace al iterador). En segundo lugar, las bases de reanudación están unidas conjuntamente por medio de un enlace de reanudación que constituye la cadena de iteradores llamados por el módulo, tal como veíamos anteriormente. Por último, el valor cedido se asigna (por referencia) a la variable de control del bucle `for`, transfiriéndose el control al punto de retorno normal especificado en el registro de activation del iterador.

La cadena del iterador es una estructura de tipo LIFO. Una vez ejecutado el cuerpo de un bucle, la base de reanudación del iterador a reanudar será la última que se metió en la cadena LIFO del registro de activation en curso. Cada base de reanudación se saca de la cadena utilizando su información para reanudar el iterador.

Por último, el regreso desde un iterador es idéntico al regreso desde un subprograma: se saca de la pila el registro de activation del iterador y se cede el control al punto de retorno final especificado en dicho registro.

Estos mecanismos se ilustran en el siguiente ejemplo: Supóngase que un módulo M contiene el siguiente anidamiento de bucles:

```

for ... in iter1 ... do
    for ... in iter2 ... do

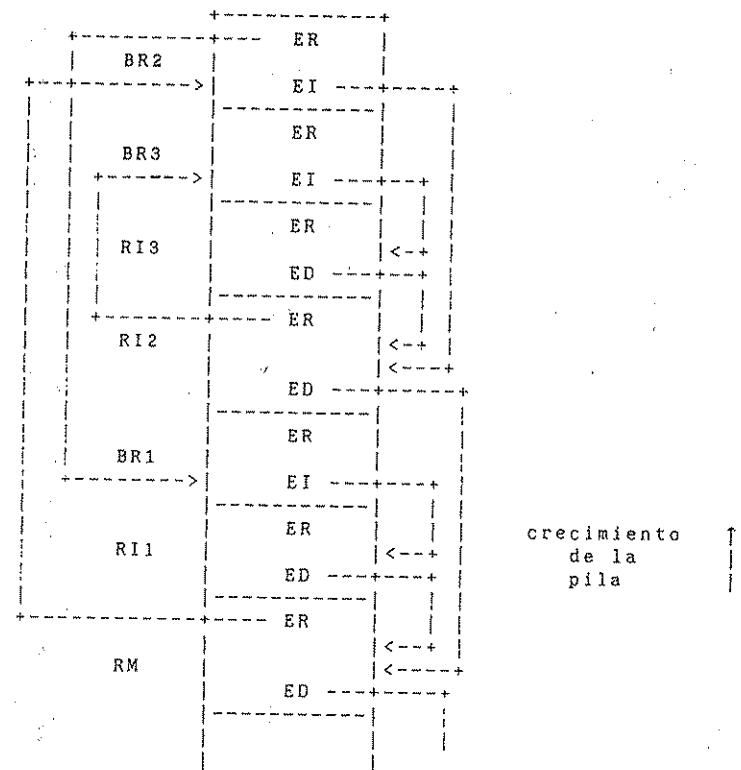
```

Supóngase también que el módulo `iter2` contiene el bucle siguiente:

```

for ... in iter3 ... do

```



Leyenda:
 RM = registro de activación del módulo M
 RII = " " " " " del iterador i
 BRI = base de reanudación " "
 EI = enlace del iterador "
 ED = enlace dinámico "
 ER = enlace de reanudación "

Figura 5.2 Configuración de la pila de ejecución con presencia de iteradores.

Cuando M llama a iter1, se mete en la cima de la pila el registro de activación RII. Después se mete la base de reanudación BRI y el enlace de reanudación del registro de activación de M (RM) que apunta a BRI. Una llamada a iter2, provocada por la ejecución del bucle más interno de M, mete en la cima de la pila el registro RI2. La posterior llamada de iter2 a iter3, seguida por la cesión desde iter3, mete en la cima de la pila RI3 y BR3, apuntando a BR2 el enlace reanudador desde RI2.

Por último, la cesión a M desde iter2 mete BR2 en la pila y en la cadena de iteradores de RM a la vez. Una vez hecho esto, el enlace reanudador de RM apunta a BR2 y el de BR2 a BR1. La Figura 5.2 representa la configuración de la pila en este momento. Para simplificar, se han omitido detalles de la estructura del registro de activación y sólo se muestra la información necesaria para seguir la ejecución en la pila.

5.2 ESTRUCTURAS DE CONTROL A NIVEL DE UNIDAD

Esta sección estudia los mecanismos que tienen los lenguajes de programación para especificar el flujo de control entre unidades de programa. El mecanismo más simple lo tenemos en el bloque de ALGOL 60, el cual crea una nueva referencia de entorno y se ejecuta cuando se encuentra durante la progresión secuencial de ejecución. Otros mecanismos más potentes permiten al programador transferir el control de una unidad a otra por medio de llamadas explícitas.

En la mayoría de los casos, la unidad llamada está subordinada a la llamante. En otras palabras, la llamante nombra explícitamente a su unidad subordinada, mientras que cada unidad sencillamente devuelve el control a la llamante, esto es, a una unidad implícita. Hay casos en los que la unidad llamada también es implícita, como ocurre con los manejadores de excepciones: una unidad puede provocar una excepción y activar explícitamente un manejador de excepciones concreto para esa excepción. También pueden organizarse las unidades en un esquema simétrico como un conjunto de correntinas, en cuyo caso dos unidades explícitas activan a una tercera. Las unidades prosiguen de una forma compartida. Por último, se pueden organizar las unidades como un conjunto de unidades concurrentes (o procesos paralelos). Con procesos concurrentes no se sabe quien cede a quien el control entre dos unidades, considerándose cada una como una unidad autónoma.

En la siguiente sección se estudian las estructuras de control según el esquema de clasificación anterior.

5.2.1 Llamadas Explícitas a Unidades Subordinadas

Este tipo de llamadas comprende el caso de los subprogramas, desde las subrutinas y funciones de FORTRAN hasta los procedimientos de Ada. Las características básicas en tiempo de ejecución de la activación y retorno de las unidades se han visto en el Capítulo 3. En esta sección vamos a tratar en particular un aspecto importante omitido deliberadamente en el capítulo citado: el paso de parámetros a los subprogramas.

El paso de parámetros posibilita la comunicación de datos entre unidades de programa. A diferencia de la comunicación a través de entornos globales, los parámetros permiten transferir datos diferentes en cada llamada y proporcionan ventajas en

términos de legibilidad y modificabilidad.

Muchos lenguajes de programación utilizan un método posicional en las llamadas a subprogramas para hacer corresponder los parámetros reales con los parámetros formales. Si un procedimiento se declara como

```
subprogram S(F1,F2,...Fn);
```

```
:  
:  
end S
```

y la llamada al subprograma es

```
call S(R1,R2,...Rn)
```

entonces el método posicional implica que el parámetro posicional F_i corresponde al parámetro real R_i , $i=1,2,\dots,n$. El método posicional tiene algunas desventajas en términos de legibilidad, cuando la lista de parámetros es grande. Una forma alternativa (opcional en Ada) es la que se conoce como el método de paso de parámetros por nombre. En las llamadas se indica explícitamente la correspondencia entre los parámetros reales y formales. Por ejemplo, veamos como sería la llamada (en pseudocódigo) a un subprograma inicializar cuyos parámetros formales tabla y valor sirven para inicializar al valor valor todos los elementos de la matriz entera tabla:

```
inicializar(tabla es x, valor es v)
```

donde x y v son los parámetros reales que corresponden a tabla y valor respectivamente.

El método por nombre es particularmente útil si el lenguaje permite especificar correspondencias por defecto con los parámetros formales. En tal caso, en una llamada a un subprograma se puede especificar sólo una parte de los parámetros reales. En el ejemplo anterior, si al comienzo del subprograma se indica 0 como valor por defecto de v , para iniciar a ceros la matriz b se haría

```
inicializar(tabla is b)
```

Podemos diferenciar tres clases de entidades que pueden pasarse como parámetros: datos, tipos y subprogramas. Pasar un tipo a un procedimiento genérico tiene las implicaciones descritas en la Sección 4.7.3 para tipos abstractos de datos genéricos. A continuación veremos los datos como parámetros, y en la Sección 5.2.1.2, los subprogramas.

5.2.1.1 Datos como Parámetros

Hay diferentes sistemas para el paso de parámetros a subprogramas. Es preciso conocer el sistema adoptado por un lenguaje, ya que la elección afecta a la semántica de dicho lenguaje. Un mismo programa puede producir diferentes resultados bajo diferentes sistemas de paso de parámetros. (Véase el Ejercicio 5.5 como ejemplo).

En las Secciones 5.2.1.1.1 a 5.2.1.1.3 veremos tres sistemas para el paso de parámetros.

5.2.1.1.1 Llamada por Referencia

La unidad que llama pasa a la unidad llamada la dirección del parámetro real (que está en el entorno de la unidad llamante). Una referencia al correspondiente parámetro formal en la unidad llamada se trata como una referencia a la posición de memoria cuya dirección se ha pasado. Entonces, una variable pasada como parámetro real es compartida, es decir, modificable directamente por el subprograma. Si el parámetro real es una expresión, el subprograma recibe en el registro de activación de la unidad llamante la dirección de la posición temporal que contiene el valor de la expresión.

En términos del modelo descrito en el Capítulo 3, las direcciones de los parámetros pueden almacenarse en posiciones específicas del registro de activación de la unidad llamante, y las referencias a esos parámetros en la unidad llamada pueden tratarse como referencias indirectas a través de esas posiciones. Puesto que un subprograma tiene un número fijo de parámetros, la cantidad de posiciones y sus equivalentes se fijan estáticamente.

5.2.1.1.2 Llamada por Copia

A diferencia de la llamada por referencia, en la llamada por copia los parámetros formales no comparten la memoria con los parámetros reales; es más, actúan como variables locales. Por tanto, la llamada por copia protege a la unidad llamante de posibles modificaciones de los parámetros reales. Por otra parte, podemos clasificar la llamada por copia en tres tipos, dependiendo de la forma en que se inicializan las variables locales correspondientes a los parámetros formales y de la forma en que finalmente afecten sus valores a los parámetros reales. Estos tres tipos se denominan por valor, por resultado y por valor-resultado.

En la llamada por valor, la unidad llamante evalúa los parámetros reales y los utiliza para inicializar los parámetros formales, los cuales actúan como variables locales en la unidad llamada. La llamada por valor no permite la devolución de información al llamante, ya que las asignaciones a los parámetros formales no afectan a la unidad llamante.

En la llamada por resultado, las variables locales correspondientes a los parámetros formales no se ponen en la llamada al subprograma pero su valor, al acabar, se devuelve a la posición de los parámetros reales en el entorno del llamante. La llamada por resultado no permite enviar información a la unidad llamada.

En la llamada por valor-resultado, las variables indicadas por los parámetros formales se inicializan en la llamada al subprograma (como en la llamada por valor) y se devuelven al finalizar (como en la llamada por resultado).

5.2.1.1.3 Llamada por Nombre

Al igual que en la llamada por referencia, un parámetro formal, antes que ser una variable local del subprograma, indica una posición en el entorno del llamante. Sin embargo, a diferencia de la llamada por referencia, los parámetros formales no están limitados a una posición en el momento de la llamada; es más, están limitados a una posición (posiblemente diferente) cada vez que son utilizados por el subprograma. Por consiguiente, cada asignación a un parámetro formal puede referirse a posiciones diferentes. Básicamente, en la llamada por nombre cada parámetro formal se reemplaza textualmente por el parámetro real. Esta regla, aparentemente simple, puede llevarnos a complicaciones insospechadas. Por ejemplo, veamos el siguiente procedimiento, en el cual se intenta intercambiar los valores de a y b (a y b son parámetros por nombre):

```
procedure cambio(a,b: integer);
var auxiliar: integer;
begin
    auxiliar := a;
    a := b;
    b := auxiliar
end cambio;
```

Obtendremos un resultado imprevisible e incorrecto con la llamada

```
cambio(i, a[i])
```

ya que la regla de sustitución especifica que las sentencias a ejecutar son

```
auxiliar := i;
i := a[i];
a[i] := auxiliar
```

Si antes de la llamada, i=3 y a[3]=4, después de la llamada tendremos i=4 y a[4]=3, quedando a[3] como estaba.

Otro resultado incorrecto se obtiene cuando el parámetro real que es sustituido (conceptualmente) en la unidad llamada,

pertenece al entorno de referencia del llamante en vez de a la activación de la unidad llamada. Por ejemplo, supongamos que el procedimiento cambio también cuenta el número de veces que es llamado, y que a su vez está incluido en el procedimiento siguiente:

```
procedure x...
var c: integer;
procedure cambio(a,b: integer);
var auxiliar: integer;
begin
    auxiliar := a;
    a := b;
    b := auxiliar;
    c := c+1
end cambio;
procedure y...
var c,d: integer;
...
cambio(c,d);
...
end y;
...
end x;
```

Cuando y llama a cambio la regla de sustitución especifica que las sentencias a ejecutar son:

```
auxiliar := c;
c := d;
d := auxiliar;
c := c+1
```

Sin embargo, la posición ligada al nombre c en la última sentencia pertenece al registro de activación del procedimiento x, mientras que la posición, ligada a c en anteriores referencias pertenece al registro de activación de y. Obsérvese que el problema no está en la identificación del parámetro real, ya que puede hacerse fácilmente en tiempo de traducción. El problema está en la dificultad por parte del programador de prever en tiempo de ejecución la correspondencia entre los parámetros reales y formales.

Por lo tanto, el uso de la llamada por nombre nos lleva probablemente a programas difíciles de leer así como difíciles de implementar. La técnica básica de implementación consiste en sustituir cada referencia a un parámetro formal por una llamada a un subprograma (denominado usualmente THUNK) que evalúa una referencia a un parámetro real y su entorno apropiado. Se crea el THUNK por cada parámetro real. Evidentemente, el paso de las llamadas a THUNK en tiempo de ejecución hacen muy costosa la llamada por nombre.

La llamada por referencia es el sistema estándar utilizado por FORTRAN para pasar parámetros. La llamada por nombre es

estándar en ALGOL 60, pero el programador puede utilizar opcionalmente la llamada por valor. SIMULA 67 proporciona las llamadas por valor, por referencia y por nombre.

Pascal permite pasar parámetros bien por valor, bien por referencia. Por ejemplo, la cabecera del procedimiento

```
procedure x(y:integer; var z:tipousuario)
```

especifica que y se pasa por valor, mientras que z (de tipo tipousuario) se pasa por referencia (indicado por la palabra reservada var). La elección entre uno u otro sistema puede venir dictada por diversas consideraciones, como el evitar efectos laterales no deseados provocados por modificaciones inadvertidas de parámetros formales o por eficiencia de implementación. En general, el paso de parámetros por referencia es más costoso, en términos de tiempo de proceso, si los parámetros formales se utilizan con frecuencia en el subprograma llamado, ya que cada acceso requiere direccionamiento indirecto. Sin embargo, si los objetos son grandes y/o sólo se accede a ellos unas pocas veces, la llamada por copia será costosa tanto en memoria como en tiempo de proceso.

ALGOL 68 dispone de un solo mecanismo para el paso de parámetros, similar a la llamada por valor. Sin embargo, un parámetro puede ser de tipo ref m, es decir, su valor es una referencia a un objeto de tipo m. Por lo tanto, es posible obtener también el efecto de la llamada por referencia.

Concretamente, el significado de la llamada

```
p(i)
al procedimiento
proc p = (int a) void;
  (... cuerpo ...)
es lo mismo que
(int a = i;
  (... cuerpo ...))
```

Por lo tanto, a toma el valor de i, es decir, actúa como una constante entera dentro del ámbito de la llamada. Así, el parámetro es casi como la llamada por valor, excepto que no puede cambiarse. Supongamos ahora que el procedimiento se ha declarado como sigue:

```
proc p = (ref int a) void;
  (... cuerpo ...)
```

entonces, la llamada p(i) se expandiría (conceptualmente) en:

```
(ref int a = i;
  (... cuerpo ...))
```

que permite al procedimiento acceder al parámetro real i a través de la referencia a, es decir, exactamente una llamada por referencia. Se explotan pues la variedad de estructuras y la ortogonalidad de ALGOL 68 para obtener un mecanismo de paso de parámetros uniforme.

La llamada por resultado puede formar lo que se denomina subprograma de tipo función, proporcionado por lenguajes como Pascal, donde el nombre del subprograma función actúa como una variable local que contiene al final el valor producido por el propio subprograma función.

Ada dispone de tres sistemas para el paso de parámetros: in, out e in out. Se asume in por defecto. Un parámetro formal in actúa como una constante local cuyo valor lo proporciona el correspondiente parámetro real y no puede ser modificado por el subprograma. Un parámetro out actúa como una variable local cuyo valor se asigna al correspondiente parámetro real al salir del subprograma. Un parámetro formal in out actúa como una variable local y permite el acceso y la asignación al correspondiente parámetro real. La definición de Ada no indica si el paso de parámetros debe implementarse por referencia o por copia. La elección entre ambas implementaciones se deja por entero al traductor, el cual elige en base a la eficiencia necesaria. Se consideran erróneos los programas de Ada que producen resultados diferentes según la implementación utilizada. Sin embargo, el traductor no puede, por desgracia, detectar programas erróneos y por lo tanto puede producir resultados diferentes cuando se cambia de instalación. La razón para la posible discrepancia entre ambas instalaciones es que en el paso de parámetros por copia, la variable local asociada al parámetro formal ocupa una nueva posición de memoria. Por otra parte, la llamada por referencia crea una nueva vía de acceso a una posición de memoria que puede ser accedida por el mismo subprograma por medio de otras vías (por ejemplo, un nombre global). Esta cuestión se estudiará más a fondo en la Sección 6.2.

5.2.1.2 Subprogramas como Parámetros

Es útil pasar subprogramas como parámetros en muchas situaciones prácticas. Por ejemplo, se puede escribir, sin conocer la función, un subprograma S que evalúa propiedades analíticas de las funciones de una variable en un intervalo dado a..b, y poder usarlo para diferentes funciones, si los valores de la función los produce otro subprograma que se envía a S como parámetro. Otro ejemplo puede ser el caso de un lenguaje que no dispone de características explícitas para el manejo de excepciones, en el cual podemos transmitir como parámetro el subprograma manejador de excepciones a la unidad que pueda producir la excepción.

Como veremos a continuación, los subprogramas como parámetros, aunque útiles, facilitan la creación de programas oscuros debido a la dificultad de conocer sus entornos de referencia. Para simplificar, nos restringiremos a aquellos lenguajes con reglas estáticas de ámbito, como ALGOL.

El envío de un subprograma como parámetro requiere pasar al menos una referencia al segmento de código del parámetro real. Sin embargo, el subprograma pasado puede acceder también a variables no locales, siendo necesario entonces pasar información acerca del entorno no local del subprograma parámetro real. En los lenguajes tipo ALGOL, el entorno no local accesible por un subprograma P, enviado como parámetro en una llamada de tipo X(P), es el limitado a P cuando X(P) se activa. En otras palabras, si Y es el programa que hace la llamada X(P), el entorno no local asociado con el parámetro P se determina por la cadena estática originada en el registro de activación que contiene una definición de P, y que es visible cuando se ejecuta Y. Por consiguiente, los subprogramas parámetros se pueden representar por un par ordenado: un puntero pc al segmento de código y un puntero pr al registro de activación de la unidad que contiene la definición del subprograma. Una llamada al subprograma parámetro requiere llevar el registro de activación a la cima de la pila, poner el enlace estático del valor de pr y transferir el control al código especificado por pc.

Por ejemplo, considérese el siguiente fragmento:

```
procedure principal...
  ...
  procedure a...
    ...
  end a;
  procedure b (procedure x);
  var y: integer;
    procedure c...
      ...
      y := ...;
      ...
    end c;
    x;
    b(c);
  ...
end b;
...
b(a);
...
end principal;
```

El procedimiento principal llama a b con el procedimiento a como parámetro. La llamada a x, emitida por el procedimiento b, activa el procedimiento a. Entonces, el procedimiento b se llama recursivamente con el procedimiento c como parámetro. En las Figuras 5.3.a y 5.3.b respectivamente se muestra la pila antes y después de esta llamada. (Para simplificar, los nombres de las

variables se usan en el registro de activación, aunque no es necesario, tal como se vió en la Sección 3.6). En este punto, la ejecución de una nueva llamada a x activa al procedimiento c (Figura 5.3.c). Este ejemplo muestra que cuando en un programa se usan los procedimientos como parámetros, las variables no locales visibles en un punto dado de la ejecución no son necesariamente aquellas pertenecientes al último registro de activación de la unidad en la cual fueron declaradas. Por ejemplo, la asignación a Y realizada por el procedimiento c modifica el contenido de una posición de memoria que no está en el registro de activación de b, sino en el siguiente (Figura 5.3.c).

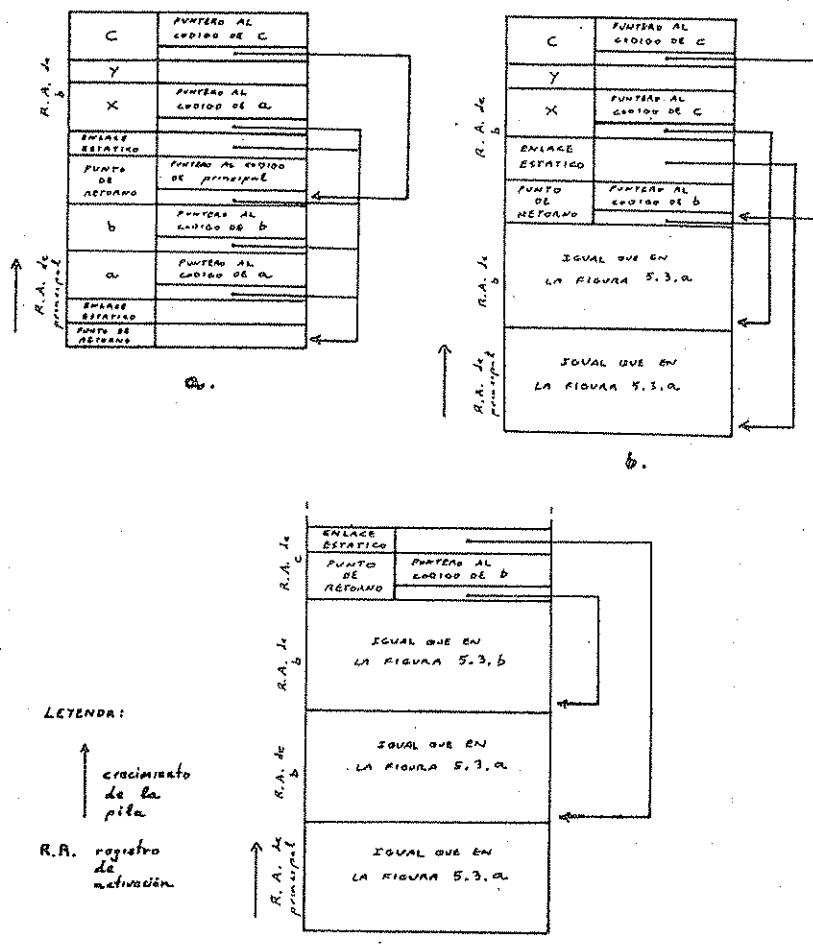


Figura 5.3 Evolución de la pila de ejecución con el uso de procedimientos como parámetros.

5.2.2 Llamadas Implicitas a Unidades

Esta sección analiza las unidades que se llaman implicitamente cuando se alcanza una condición de excepción. Difícilmente se puede especificar el concepto de "condición de excepción". Se desea que los programas se comporten "razonablemente" bajo un amplio abanico de circunstancias, incluso en presencia de fallos internos de hardware/software, o de hechos que ocurren tan pocas veces que no se consideran anómalos, o de datos de entrada no válidos. Sin embargo, compete al diseñador considerar qué es un estado de proceso "normal" y dependerá en último lugar de la naturaleza de la aplicación. Como consecuencia, un estado de proceso "anómalo" (o excepción) no significa necesariamente que estamos ante un error "catastrófico", sino más bien que la unidad que se ejecuta está incapacitada para proceder de una manera tal que se llegue a una terminación normal. La incapacidad del programa para detectar estados anómalos de proceso y manejarlos de la forma apropiada puede producir un software que trabaje sólo bajo ciertas circunstancias, es decir, software no tolerante con los fallos.

Los lenguajes tradicionales (excepto PL/I) no ofrecen especial ayuda para el apropiado manejo de estados de proceso anómalos. Sin embargo, un cierto número de lenguajes recientes proporcionan características ad hoc para el manejo de excepciones. Con estas características, lo concerniente a anomalías puede sacarse fuera del flujo principal del programa, a fin de no oscurecer el algoritmo básico.

Las principales cuestiones que se suscitan en el manejo de excepciones son:

- 1.- ¿Cómo se declara una excepción y cuál es su ámbito?
- 2.- ¿Cómo se alcanza una excepción?
- 3.- ¿Cómo podemos especificar las unidades (manejadores de excepciones) que se han de ejecutar cuando se alcanzan las excepciones?
- 4.- ¿Cómo se limita una excepción a un manejador?
- 5.- ¿A dónde se cede el control una vez tratada una excepción?

Veamos primero el punto 5, ya que su solución tiene un fuerte impacto en la potencia y utilidad del mecanismo. La elección básica en el diseño del lenguaje es si después de la ejecución del manejador de excepciones apropiado, se permite la devolución del control al punto donde surgió la excepción. En tal caso, el manejador de excepciones puede realizar sus acciones de corrección y al terminar, puede continuar la ejecución normal. PL/I y Mesa han adoptado este punto de vista. El mecanismo resultante es flexible y potente, pero es difícil de dominar por

programadores inexpertos. De hecho, facilita la programación insegura quitando el síntoma de un error sin quitar la causa de dicho error. Por ejemplo, se puede tratar la excepción provocada por un valor erróneo de un operando generando arbitrariamente un valor aceptable.

La otra solución consiste en terminar la ejecución de la unidad que alcanza la excepción y transferir el control al manejador de excepciones. Conceptualmente, significa que no puede reanudarse la acción que alcanzó la excepción. Desde el punto de vista de la implementación, significa que puede borrarse el registro de activación de la unidad afectada. Bliss, CLU y Ada han adoptado este esquema más sencillo.

Veamos ahora una reseña de las características del manejo de excepciones que ofrecen algunos lenguajes.

5.2.2.1 Manejo de Excepciones en PL/I

Las excepciones de PL/I se denominan condiciones (CONDITIONS). Los manejadores de excepciones se declaran con sentencias ON:

`ON CONDITION (nombre_excepción) manejador_de_excepción`

donde `manejador_de_excepción` puede ser una sentencia simple o un bloque. Se alcanza explícitamente una excepción con la sentencia

`SIGNAL CONDITION (nombre_excepción);`

El lenguaje define también un número de excepciones incorporadas y proporciona manejadores definidos por el sistema para ellas. Las excepciones incorporadas se alcanzan automáticamente en la ejecución de algunas sentencias (p.ej., se alcanza ZERODIVIDE cuando en la evaluación de una expresión, el denominador de una división se encuentra a cero). La acción realizada por un manejador del sistema la especifica el lenguaje. Sin embargo, esa acción puede ser redefinida como una excepción del usuario:

`ON ZERODIVIDE BEGIN;`

`...`

`END;`

Cuando en el transcurso de la ejecución encontramos una unidad ON, se establece un nuevo enlace entre una excepción y un manejador. Una vez establecida la asociación, permanece válida hasta que es anulada por la ejecución para la misma excepción de otra sentencia ON, o hasta que acabe el bloque al que pertenece la sentencia ON. Si en el mismo bloque hay varias sentencias ON para la misma excepción, cada nuevo enlace anula al anterior. Si aparece una sentencia ON para la misma excepción en un bloque interno, el nuevo enlace permanecerá vigente sólo hasta que acabe

el bloque interno. Cuando salimos de un bloque, se restauran los enlaces que existían en la entrada del bloque anterior.

Cuando se alcanza una excepción (bien automáticamente, bien por una sentencia SIGNAL) se ejecuta el manejador vinculado actualmente a la excepción como si fuera un subprograma llamado explícitamente desde ese punto. Por lo tanto, a no ser que se especifique lo contrario en el manejador, el control volverá al punto en que se produjo la sentencia SIGNAL.

Hemos visto desde el principio que es sumamente dinámico el enlace entre una excepción que se alcanza en un cierto punto del programa y el manejador al que se recurre. Por consiguiente, los programas en PL/I que utilizan esta construcción pueden ser poco manejables, o difíciles de escribir y comprender. Además, el lenguaje no permite pasar parámetros desde el punto donde se produce la excepción al correspondiente manejador. Por consiguiente, sólo se puede establecer el flujo de información mediante el uso de variables globales, lo que por otro lado puede ser una práctica poco recomendable. Además, no siempre es posible utilizar variables globales. Por ejemplo, cuando se alcanza la excepción STRINGRANGE, que indica un intento de acceder más allá del límite de una cadena de caracteres, no hay manera de que el manejador sepa a qué cadena se refiere si hay dos o más cadenas visibles en el ámbito. Tales situaciones hacen que el manejo de excepciones en PL/I sea a menudo ineficaz.

Los mecanismos de manejo de excepción en PL/I pueden complicarse más con la habilitación e inhabilitación de las excepciones incorporadas; las excepciones definidas por el usuario no se pueden inhabilitar, ya que de todos modos se deben indicar explícitamente. Muchas de las excepciones incorporadas están habilitadas por defecto y vinculadas al manejo de errores del sistema. La habilitación de una excepción previamente inhabilitada (o una excepción que no está habilitada por defecto) se hace anteponiendo una sentencia, bloque o procedimiento con el nombre de la excepción. Por ejemplo,

```
(ZERODIVIDE): BEGIN;
  ...
END;
```

El ámbito del prefijo es estático, es decir, es la sentencia, bloque o procedimiento al que está ligado. Una excepción habilitada puede inhabilitarse explícitamente con anteponer NOnombre_excepción a la sentencia, bloque o procedimiento en cuestión. Por ejemplo,

```
(NOZERODIVIDE): BEGIN;
  ...
END;
```

5.2.2.1.1 Modelo de Implementación

Veamos someramente una implementación del mecanismo de manejo de excepciones en PL/I. Cuando se encuentra una unidad ON durante la ejecución, se salvan el nombre de la condición y su referencia al manejador en una entrada del registro de activación de la unidad en curso. Todas las entradas de unidades ON constituidas para una unidad dada de activación U, son accesibles por medio de una posición fija del registro de activación de U. Cuando se alcanza una condición, se hace una búsqueda entre las entradas de unidades ON, empezando por la más reciente, hasta que se encuentra la que corresponde a la condición dada. Si no se encuentra ninguna, se toma la acción correspondiente por defecto. La principal desventaja de esta implementación es que la búsqueda puede llevar una cantidad de tiempo significativa. No obstante, la búsqueda sólo tiene lugar cuando se alcanza una excepción. Se puede aumentar la eficiencia de la búsqueda si a cada bloque se le proporciona un acceso más directo a cada unidad ON que le afecte. Concretamente, se podría tener una tabla con todas las condiciones que pueda alcanzar un programa. La entrada para una condición C es un puntero a la pila de punteros que apuntan a los registros de activación de las unidades ON para C. Inicialmente, todas las pilas están vacías. Si durante la ejecución se encuentra una unidad ON, se pone una nueva entrada en la cima de la pila correspondiente. Cuando termina un bloque de activación, se saca de las pilas todo puesto durante la activación.

5.2.2.2 Manejo de Excepciones en CLU

El manejo de excepciones en CLU es menos potente que en PL/I, pero es más fácil de utilizar y parece proporcionar formas efectivas para el manejo de situaciones anómalas.

Las excepciones sólo se pueden alcanzar en los procedimientos. Las excepciones que un procedimiento pueda señalar han de declararse en la cabecera del mismo. Se alcanza explícitamente una excepción por medio de la instrucción signal. Las operaciones incorporadas pueden alcanzar un conjunto conocido de excepciones; por ejemplo, una división puede señalar que el valor del denominador es cero.

Cuando se alcanza una excepción, el procedimiento devuelve el control al llamante inmediato, que es el responsable de proporcionar el manejador para la excepción. A diferencia de PL/I, el procedimiento que alcanza la excepción no puede ser reanudado, incluso después de realizar las acciones de recuperación. Además, el manejador está ligado estrictamente al punto de llamada.

El manejador de excepciones se relaciona con las sentencias por medio de la cláusula except, según la sintaxis

```
<sentencia> except <lista_de_manejadores> end
```

donde **<sentencia>** puede ser cualquier sentencia (compuesta) del lenguaje. Si la ejecución de un procedimiento que ha sido llamado en **<sentencia>** alcanza una excepción, se cede el control a **<lista_de_manejadores>**. Una **<lista_de_manejadores>** tiene el siguiente formato:

```
when <lista_de_excepcion_1>: <sentencia_1>
  ...
when <lista_de_excepcion_n>: <sentencia_n>
```

Si la excepción alcanzada pertenece a **<lista_de_excepcion_i>**, entonces se ejecuta **<sentencia_i>** (el cuerpo del manejador). Cuando acaba la ejecución del manejador, el control pasa a la sentencia siguiente a la que está ligado dicho manejador. Si **<sentencia_i>** contiene una llamada a una unidad, pudiera ser que se alcanzara otra excepción. En tal caso, el control se pasa a la sentencia **except** que incluye **<sentencia>**. Si la excepción alcanzada no figura en ninguna lista de excepciones, se repite el proceso para las sentencias incluidas estáticamente. En caso de no encontrar un manejador en el procedimiento que hizo la llamada, el procedimiento señala implícitamente a la excepción predefinida **failure** y devuelve el control.

Otra diferencia notable frente a PL/I es que las excepciones pueden devolver parámetros a sus manejadores, con el consiguiente ahorro de variables globales. Un ejemplo típico es la excepción **failure**, la cual devuelve una cadena de caracteres que denotan una excepción no tratada por el manejador.

Deliberadamente, el diseño de CLU no permite mecanismos para inhabilitar las excepciones. El motivo es que, a menos que se sepa que una excepción no va a ocurrir, sería peligroso inhabilitarla. Además, la inhabilitación de excepciones poco probables la puede hacer un compilador con optimizador de código, lo cual evitaría por una parte el esfuerzo de detectar excepciones improbables y por otra, el espacio ocupado por los manejadores.

5.2.2.2.1 Modelo de Implementación

La implementación del manejo de excepciones en CLU es más bien sencilla: cuando se señala una excepción, se devuelve el control a su llamante como si fuera un retorno normal, con la única diferencia de que el punto de retorno no es la instrucción siguiente a la de llamada, sino el manejador de excepciones más inmediato.

La implementación de CLU asocia cada procedimiento con una tabla de manejadores (de contenido fijo) que almacena información de todos los manejadores del procedimiento. Cada entrada de la tabla contiene:

- a. El conjunto de excepciones gobernadas por el manejador.
- b. Dos punteros a la porción de texto del procedimiento llamante al que está ligado el manejador (el ámbito del manejador).
- c. Un puntero al código del manejador.

Cuando se produce una excepción en un procedimiento P, la dirección de la instrucción que produjo la llamada se halla en el registro de activación del procedimiento llamante y se utiliza ese valor para encontrar el punto de retorno apropiado mediante búsqueda en la tabla de manejadores de P.

5.2.2.3 Manejo de Excepciones en Ada

El manejo de excepciones en Ada es similar al utilizado en Bliss y Gypsy. Asimismo, también tiene semejanzas con el de CLU.

En un programa se puede provocar explícitamente una excepción con la sentencia **raise**, por ejemplo:

```
raise AYUDA
```

Los manejadores de excepciones están ligados al cuerpo de un subprograma o un "package", o a un bloque con la palabra reservada **exception**. Por ejemplo,

```
begin...
exception when AYUDA =>...
  when DESESPERADO =>...
end;
```

Si la unidad que alcanza una excepción proporciona un manejador para la misma, se trasfiere inmediatamente el control al manejador; se saltan las acciones que siguen al punto donde se alcanzó la excepción, se ejecuta el manejador y termina la unidad. Si la unidad que se está ejecutando, U, no proporciona un manejador, se propaga la excepción. Si U es un bloque, termina su ejecución y se vuelve a alcanzar implícitamente la excepción en la unidad que engloba a U. Si U es un cuerpo de un subprograma, se hace el retorno desde el mismo y se realanza implícitamente la excepción en el punto de llamada. Si U es el cuerpo de un "package", actúa como si fuera un procedimiento que se activa cuando se procesa la declaración del "package". En el caso de que no haya un manejador asociado al "package", se propaga la excepción a la unidad que alberga a dicho "package".

El mecanismo de manejo de excepciones en Ada es multinivel, frente al de CLU que es de nivel único. En otras palabras, una excepción de Ada alcanzada en una unidad dada puede ser manejada por otras unidades de su llamador inmediato. Puede justificarse la restricción de CLU en base a la programación metódica. De hecho, CLU ve un procedimiento como la implementación de una

acción abstracta. El que llama a un procedimiento quiere que dicho procedimiento ejecute su acción abstracta, y no precisa conocer de la forma que lo hace. Las excepciones que puede alcanzar un procedimiento caracterizan el comportamiento abstracto del mismo y ha de conocerlas el llamador. Sin embargo, no necesita saber nada acerca de las excepciones que puedan alcanzar los procedimientos locales usados para la implementación de la acción abstracta. El mismo razonamiento sirve también para explicar por qué, si surge una excepción, un procedimiento CLU debe hacer retorno, y además no se puede recuperar. Manejar las excepciones de forma que permitan continuar a la unidad llamada requiere que el llamador conozca la estructura interna de la unidad llamada.

CLU, también en contraste con Ada, no permite que una unidad maneje por sí misma la excepción que ha alcanzado. La justificación es que, por definición, las excepciones son condiciones anómalas señaladas por un procedimiento, el cual es incapaz de manejarlas. No obstante, la alternativa de Ada parece más idónea, ya que permite al programador especificar fácilmente algunas acciones de "limpieza" que pueda ejecutar la unidad llamada antes de devolver el control. CLU precisa que se devuelvan los parámetros al manejador del llamador con el fin de permitirle hacer "limpieza". Quizás sea ésta la razón por la que Ada no permite pasar parámetros a los manejadores de excepción.

5.2.2.3.1 Modelo de Implementación

La implementación del manejo de excepciones en Ada se puede realizar utilizando el concepto de tabla de manejadores de CLU. La diferencia básica es que el inmediato llamante no proporciona necesariamente el manejador de excepciones. Por tanto, podría ser necesario recorrer la cadena dinámica de llamadas a procedimientos para localizar una entrada para el manejador deseado.

5.2.3 Unidades Simétricas: Corrutinas en SIMULA 67

Se entiende por unidades simétricas (corrutinas) a un grupo de unidades de programa que activan explícitamente a otras. Tal como vimos en la Sección 2.3, cuando se transfiere el control a la unidad, su ejecución continúa en la sentencia siguiente a la que se ejecutó en último lugar. Es decir, se trata de una ejecución de unidades en forma intercalada, siguiendo una pauta predefinida.

Estudiamos en esta sección una visión simplificada del mecanismo de SIMULA 67 para activar unidades simétricas y para comutar el control desde una corrupción a otra. Una corrupción de SIMULA 67 es una class cuyo formato es el siguiente:

```
class x(parametros);
    declaraciones de parámetros;
begin
    declaraciones;
    lista_de_sentencias_1;
    detach;
    lista_de_sentencias_2
end
```

Si las variables y1 e y2 son referencias a x, podemos escribir entonces

```
y1: -new x(...); y2: -new x(...)
```

Cuando se encuentra un new, se crea una nueva class y se ejecuta su cuerpo de sentencias. Tan pronto como se detecta detach, se devuelve el control a la unidad que provocó el new. Como consecuencia de la sentencia detach, la activación de la unidad se comporta ahora como una corrupción que puede ser reanudada y a su vez puede reanudar otras corrupturas.

Veamos como ejemplo la descripción de una partida de cartas con cuatro jugadores, en la que todos usan la misma estrategia:

```
begin boolean finjuego; integer ganador;
class jugador(n, mano); integer n;
    integer array mano(1:13);
begin
    ref(jugador)siguiente;
    detach;
    while not finjuego do
begin
    realizar una jugada;
    if finjuego then ganador := n
        else resume(siguiente)
end
end;
ref(jugador)array p(1:4); integer i;
integer array cartas(1:13);
for i := 1 step 1 until 4 do
begin
    generar las cartas en la matriz para la mano del
    jugador i;
    p(i) := -new jugador(i, cartas)
end;
for i := 1 step 1 until 3 do
    p(i).siguiente: -p(i+1);
    p(4).siguiente: -p(1);
    resume p(1);
    escribir el nombre del ganador
end
```

El primer bucle del programa (for i := 1 step 1 until 4 do...) sirve para crear los cuatro jugadores que más tarde son enlazados mediante el siguiente bucle (for i := 1 step 1 until 3

do...); empezando con el jugador número 1. A partir de aquí, cada jugador da paso al siguiente según el orden establecido por los enlaces. Cuando en una jugada (que es un procedimiento que no figura en el programa) se pone finjuego a true, se asigna a ganador el nombre del jugador vencedor, terminando la corutina y devolviendo el control al programa que activó el conjunto de unidades simétricas. Por último, el programa principal escribe el nombre del ganador y acaba.

5.2.3.1 Modelo de Implementación

El ejemplo de la pila de ejecución que vimos en el Capítulo 3 no es válido para soportar corutinas. Cuando una corutina A realiza un resume a una corutina B, se debe salvar (en el registro de activación de A) el puntero a la instrucción siguiente a resume. Además, el registro de activación de A no se ha liberado. Durante la ejecución, cada corutina puede entrar a los bloques internos y a los procedimientos. Por tanto, cada corutina requiere una pila de registros de activación que puede crecer y disminuir independientemente de las otras pilas. Es decir, podemos ver la creación de un conjunto de corutinas como la creación de una nueva pila de ejecución por cada corutina. La organización de la memoria en tiempo de ejecución es pues un árbol de pilas (Figura 5.4), llamado también "pila cactus".

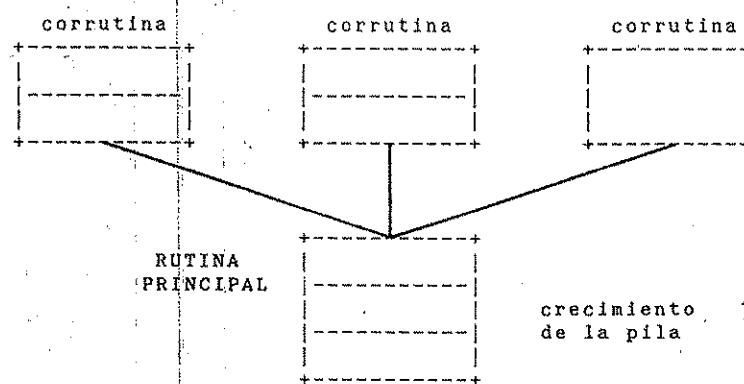


Figura 5.4 Árbol de pilas.

5.2.4 Unidades Concurrentes

Tal como vimos en la Sección 2.4, hay casos en los que el sistema que deseamos diseñar se caracteriza, abstractamente, por un conjunto de procesos que se realizan concurrentemente y que solo interactúan ocasionalmente. Se puede garantizar una interacción correcta con el uso adecuado de sentencias de

sincronización (o primitivas) proporcionadas por el lenguaje de programación. En esta sección se estudian y evalúan tres mecanismos de sincronización básicos: semáforos, monitores y rendezvous. Hemos elegido estos tres porque representan la evolución histórica del concepto de sincronización de unidades concurrentes y porque están incorporados a lenguajes que existen hoy día.

5.2.4.1 Semáforos

Los semáforos los inventó Dijkstra y posteriormente se introdujeron en ALGOL 68 como parámetros de sincronización. Un semáforo es un objeto de datos que toma valores enteros, con el que pueden operar las primitivas P y V. En ALGOL 68 se usan los nombres up y down. Cuando se declara un semáforo, se inicializa a un cierto valor entero.

Las definiciones de P y V son

```
P(semaforo): if semaforo>0 then semaforo := semaforo-1
else interrumpir el proceso en curso
```

```
V(semaforo): if hay un proceso interrumpido en el semaforo
then activar el proceso
else semaforo := semaforo+1
```

Se asume que las primitivas P y V son operaciones indivisibles, es decir, dos procesos no pueden estar ejecutando al mismo tiempo una operación P y/o V sobre un mismo semáforo. Esto ha de garantizar la implementación básica, que hará que P y V se comporten como instrucciones de máquina elementales.

El semáforo tiene (1) una estructura de datos asociada donde se registra la identidad de los procesos que esperan en el semáforo y (2) una norma para seleccionar el proceso que ha de activarse cuando se requiera una primitiva V. Normalmente, la estructura de datos es una cola de tipo FIFO (primero en entrar, primero en salir). Sin embargo, puede ser que queramos asignar prioridades a los procesos, y concebir políticas de selección más complejas basadas en tales prioridades. Veamos el ejemplo de la Sección 2.3 (productor-consumidor), resuelto con semáforos y utilizando pseudolenguaje Pascal:

```
const n=20;
shared var zona de memoria de longitud n con primitivas poner y
quitar que actualizan el campo total; es decir, el
número total de unidades almacenadas;
semaphore exclumutua := 1; (* garantiza la exclusión mutua *)
dentro := 0; (* número de unidades dentro de
la zona de memoria *)
huecos := n; (* número de sitios libres
dentro de la zona de memoria *)
```

```

Process productor;
var i: integer;
repeat
  producir(i);
  P(huecos); (* espera a que haya un sitio libre *)
  P(exclumutua); (* espera a que la zona esté libre *)
  poner i en la zona;
  V(exclumutua); (* acaba el acceso a la zona *)
  V(dentro) (* una unidad más en la zona *)
forever
end productor;
process consumidor;
var j: integer;
repeat
  P(dentro); (* espera a que haya una unidad
en la zona *)
  P(exclumutua); (* espera a que esté disponible
la zona *)
  quitar una unidad de la zona y llevarla a j;
  V(exclumutua); (* acabar el acceso a la zona *)
  V(huecos); (* un sitio libre más en la zona *)
  consumir j
forever
end consumidor

```

Las palabras reservadas process y end engloban el segmento de código que puede realizarse concurrentemente, y shared var declara la variable o variables que serán utilizadas concurrentemente por los procesos. Los semáforos huecos y dentro se utilizan para garantizar un correcto acceso a la zona de memoria. En concreto, huecos interrumpe al productor cuando intenta insertar un nuevo elemento en la zona de memoria si ésta está llena. De una forma similar, dentro interrumpe al consumidor si intenta quitar un elemento de la zona si ésta está vacía.

El semáforo exclumutua se utiliza para imponer la exclusión mutua de los accesos a la zona de memoria y es necesario dado que las operaciones poner y quitar modifican concurrentemente el valor de total. Obsérvese sin embargo que se introdujo la variable total en la Sección 2.3.2.4 con el único propósito de explicar el problema de la exclusión mutua. Se pueden implementar las operaciones poner y quitar de forma que no necesiten la exclusión mutua, proporcionando entonces (como ocurre en este caso) una seguridad sobre el correcto acceso a la zona de memoria (véase Ejercicio 5.14).

La programación con semáforos requiere la asociación de un semáforo con cada condición de sincronización. En nuestro ejemplo vemos que los semáforos son mecanismos muy sencillos pero de bajo nivel, con lo que su utilización puede ser delicada en la práctica, y los programas resultantes son a menudo difíciles de diseñar y de entender. Además, se pueden hacer pequeñas comprobaciones estáticas en los programas que utilizan semáforos. Por ejemplo, un compilador no puede detectar el uso incorrecto de un semáforo, como puede ser el resultado de cambiar V(exclumutua)

por P(exclumutua) en el proceso productor visto anteriormente (véase Ejercicio 5.12). La detección de cada uno de los errores es imposible ya que ello implicaría que el traductor conociera la semántica del programa, es decir, que las operaciones a realizar sobre la zona de memoria se realizarán con exclusión mutua y que se usa exclumutua para garantizar cada exclusión. Por tanto, los semáforos requieren una considerable autodisciplina por parte del programador. Por ejemplo, no se debe olvidar ejecutar una sentencia P antes de acceder al recurso compartido, u omitir la ejecución de V con el fin de liberar dicho recurso.

La utilización de semáforos para otra cosa que no sea la exclusión mutua, es bastante arriesgada. En el ejemplo del productor/consumidor, el consumidor se suspende a sí mismo con la ejecución de P(huecos) cuando la zona está llena. El programador no debe olvidar poner una V(huecos) después de cada acceso a la zona para consumir un dato, ya que si no, el productor quedaría bloqueado definitivamente.

PL/I fue el primer lenguaje que introdujo las unidades concurrentes llamadas tasks (tareas). Un procedimiento puede ser llamado por una tarea, en cuyo caso se ejecuta concurrentemente con el proceso que le llamó. También se pueden asignar prioridades a las tareas. Se consigue la sincronización mediante sucesos, que son similares a los semáforos pero que sólo pueden tomar dos valores binarios: "0" y "1". La realización de un suceso E se hace con la operación WAIT(E), que equivale a la operación P sobre un semáforo. La operación V del semáforo tiene su equivalente en la operación COMPLETION(E), que pone el suceso E a valor "1". PL/I amplía la noción de semáforos permitiendo una operación WAIT para diversos sucesos y para una expresión entera e. Con ello, el proceso se suspenderá hasta que se haya completado cualquier suceso e. Por ejemplo,

WAIT(E1,E2,E3)(1);

indica una espera para cualquiera de los sucesos E1, E2 o E3.

ALGOL 68 permite la descripción de procesos concurrentes en una cláusula paralela cuyas sentencias se elaboran concurrentemente. La sincronización se puede realizar mediante semáforos, que son del tipo de datos sema.

5.2.4.2 Monitores

Los monitores describen tipos abstractos de datos en un entorno concurrente. La exclusión mutua en el acceso a las operaciones que manipulan la estructura de datos se garantiza automáticamente por la implementación interna. La cooperación en el acceso a la estructura de datos compartida debe programarse explícitamente utilizando las primitivas del monitor delay y continue.

En el programa siguiente, que usa notación de Pascal

Concurrente, vemos la utilización de monitores en el ejemplo del productor/consumidor:

```

type pilafifo =
monitor
  var contenido: array[1..n] of integer; (* contenido de la
                                             zona *)
  total: 0..n;      (* número de elementos de la zona *)
  dentro:          (* posición de siguiente elemento a
                     añadir *)
  fuera: 1..n;     (* posición del siguiente elemento a
                     quitar *)
  emisor, receptor: cola;
procedure entry poner(elemento: integer);
begin
  if total=n then delay(emisor);
  contenido[dentro] := elemento;
  dentro := (dentro mod n)+1;
  total := total+1;
  continue(receptor)
end;

procedure entry quitar(var elemento: integer);
begin
  if total=0 then delay(receptor);
  elemento := contenido[fuera];
  fuera := (fuera mod n)+1;
  total := total-1;
  continue(emisor)
end;
begin
  total := 0; dentro := 1; fuera := 1
end

```

Un monitor concreto (p.ej., una zona de memoria) se puede declarar con

```

var buffer: pilafifo
y se puede crear con la sentencia
init buffer

```

La sentencia init asigna memoria para las variables englobadas dentro de la definición de monitor (p.ej., contenido, total, dentro, etc.) y ejecuta la sentencia de inicialización (que pone total a cero y dentro y fuera a uno).

El monitor define dos procedimientos, poner y quitar. Están declarados con la palabra reservada entry, la cual indica que son procedimientos que pueden ser llamados para manipular el monitor. La cooperación entre productor y consumidor se logra con el uso de las primitivas de sincronización delay y continue. La operación delay (emisor) suspende la ejecución del proceso en ejecución (p.ej., productor) y lo lleva a la cola emisor. El

proceso pierde su acceso exclusivo a la estructura de datos y es retenido hasta que otro proceso (p.ej., consumidor) ejecute la sentencia continue(emisor). De forma similar, si la zona de memoria está vacía, el proceso consumidor es retenido en la cola receptor hasta que el productor lo reanude ejecutando la sentencia continue (receptor). La ejecución de la instrucción continue la hace el proceso llamante al retornar desde la llamada al monitor. Si hay procesos esperando en la cola especificada, uno de ellos reanudará inmediatamente la ejecución del procedimiento del monitor que anteriormente había sido retenido.

Veamos a continuación la estructura de un programa en Pascal Concurrente que usa el monitor arriba citado para representar la cooperación entre el productor y el consumidor:

```

const n = 20;
type pilafifo = ... tal como antes...
type productor =
  process(memoria: pilafifo);
  var elemento: integer;
  begin cycle
    ...
    ...
    memoria.poner(elemento);
    ...
    ...
  end;
type consumidor =
  process(memoria: pilafifo);
  var dato: integer;
  begin cycle
    ...
    ...
    memoria.quitar(dato);
    ...
    ...
  end;
var miproductor: productor;
  tuconsumidor: consumidor;
  buffer: pilafifo;
begin
  init buffer, miproductor(buffer), tuconsumidor(buffer)
end

```

Los procesos no se han descrito por completo, así como las actividades cíclicas (cycle...end), etc. Se han declarado dos casos particulares (miproductor y tuconsumidor) en relación con el recurso de tipo pilafifo, y por consiguiente, activados como procesos concurrentes por la sentencia init.

Los monitores fueron planteados por Brinch Hansen y C.A.R. Hoare como mecanismos de sincronización de alto nivel, y están implementados en los lenguajes Modula y Pascal Concurrente.

5.2.4.3 Rendezvous

Rendezvous es el mecanismo que proporciona Ada para describir la sincronización de procesos concurrentes (tareas), denominados tasks en Ada. Nos vamos a centrar en las propiedades básicas del mecanismo; en aras de la simplicidad, ignoraremos otras características, tales como la interacción con otras facilidades del lenguaje.

Una tarea de Ada es una unidad de programación que se ejecuta concurrentemente con otra tarea. Su estructura es muy similar en su formato a un módulo "package". Por ejemplo, la siguiente tarea describe un proceso que maneja las operaciones poner y quitar sobre una zona de memoria:

```
task MANEJADOR_DE_BUFFER is
    entry PONER(ELEMENTO: in INTEGER);
    entry QUITAR(ELEMENTO: out INTEGER);
end;
task body MANEJADOR_DE_BUFFER is
    N: constant INTEGER := 20;
    CONTENIDO: array(1..N) of INTEGER;
    DENTRO, FUERA: INTEGER range 1..N := 1;
    TOTAL: INTEGER range 0..N := 0;
begin
    loop
        select
            when TOTAL < N =>
                accept PONER(ELEMENTO: in INTEGER) do
                    CONTENIDO(DENTRO) := ELEMENTO;
                end;
                DENTRO := (DENTRO mod N)+1;
                TOTAL := TOTAL+1;
            or
            when TOTAL > 0 =>
                accept QUITAR(ELEMENTO: out INTEGER) do
                    ELEMENTO := CONTENIDO(FUERA);
                end;
                FUERA := (FUERA mod N)+1;
                TOTAL := TOTAL-1;
        end select;
    end loop;
end MANEJADOR_DE_BUFFER;
```

La parte visible de la tarea MANEJADOR_DE_BUFFER especifica dos entradas (entry) a la tarea, PONER y QUITAR. Las entradas pueden ser llamadas por otras tareas como si fueran procedimientos. Sin embargo, y a diferencia de los procedimientos, los cuerpos de las entradas no se ejecutan nada más llamarlos, sino cuando la tarea a la que pertenece la entrada está disponible para aceptar la llamada con la ejecución de la

sentencia accept correspondiente. En este punto, podemos ver las tareas llamante y llamada como reunidas en un rendezvous. Si la tarea llamante provoca la llamada antes de que la tarea llamada ejecute un accept, la tarea llamante se suspende hasta que ocurre el rendezvous. De forma similar, la tarea llamada se suspenderá si se ejecuta una sentencia accept antes de la llamada correspondiente. Obsérvese que una tarea puede aceptar llamadas desde más de una tarea; por consiguiente, cada entrada tiene potencialmente una cola de tareas llamantes.

La sentencia accept es similar a un procedimiento. Después de la repetición de la cabecera de las entradas, el cuerpo del accept (do...end) especifica las sentencias que se ejecutarán en el rendezvous. Una vez que se emparejan la llamada a una entrada con su correspondiente accept, la tarea llamante queda suspendida hasta que la tarea llamada ejecute el cuerpo del accept. Los posibles parámetros out (como en el caso de QUITAR) se devuelven a la llamante al finalizar el rendezvous, es decir, cuando se completa la ejecución del cuerpo del accept. A continuación, las dos tareas que se juntaron en rendezvous pueden continuar en paralelo.

Veamos, en el ejemplo del productor/consumidor, los cuerpos de las tareas PRODUCTOR y CONSUMIDOR, las cuales interactúan con el MANEJADOR_DE_BUFFER:

```
PRODUCTOR
loop
    producir un nuevo valor V;
    MANEJADOR_DE_BUFFER.PONER(V);
    exit when V indica el fin de los datos;
end loop;

CONSUMIDOR
loop
    MANEJADOR_DE_BUFFER.QUITAR(V);
    consumir V;
    exit when V indica el fin de los datos;
end loop;
```

En el ejemplo del MANEJADOR_DE_BUFFER, las sentencias accept están englobadas dentro de una sentencia select, la cual es similar a la sentencia if con guarda del lenguaje de Dijkstra (Sección 5.1.2) y especifica diversas alternativas separadas por or, que se pueden elegir de una forma indeterminada. La selección en Ada se especifica con la sentencia accept, la cual puede ir precedida (como en nuestro caso) por when condición. Veamos a continuación cómo es la ejecución de la sentencia de selección (select): (*)

(*) Omitimos algunas características de Ada con el fin de no complicar la implementación (y la explicación).

- a. Se evalúan todas las condiciones `when` de todas las alternativas. Las que tienen condición a valor verdadero, o simplemente no tienen opción `when`, se consideran abiertas; en caso contrario, cerradas. En el ejemplo, ambas alternativas estaban abiertas si $0 < TOTAL < N$.
- b. Se puede seleccionar una alternativa abierta si es posible el rendezvous (es decir, si una tarea ha realizado ya la llamada a la entrada). Una vez seleccionada, se ejecuta el cuerpo del correspondiente `accept`.
- c. Si hay alternativas abiertas pero de momento no se puede seleccionar ninguna, la tarea queda en espera hasta que sea posible el rendezvous.
- d. Si no hay alternativas abiertas, se señala una condición de error con la excepción definida por el lenguaje `SELECT_ERROR`.

5.2.4.4 Análisis Comparativo

Los semáforos, monitores y rendezvous son primitivas para sistemas concurrentes. Hemos señalado ya que los semáforos son mecanismos más bien de bajo nivel: los programas son difíciles de leer y escribir y no hay un control automático sobre su correcto uso. Por otro lado, los monitores son mecanismos de estructuración de alto nivel. En un lenguaje como el Pascal concurrente, el sistema de estructuración se realiza: (1) por identificación de los recursos compartidos como objetos abstractos con las oportunas primitivas de acceso, y (2) por procesos que cooperan durante el uso de los recursos que están encapsulados dentro de los monitores. La implementación garantiza automáticamente la exclusión mutua en el acceso a recursos compartidos, mientras que ha de obligarse explícitamente la cooperación suspendiendo y activando procesos por medio de las sentencias `delay` y `continue`.

En Ada desaparece la distinción entre entidades de multiprogramación activas y pasivas (procesos y monitores respectivamente). Los recursos a compartir se representan en Ada mediante tareas, es decir, componentes activos que representan los administradores de recursos. Una petición para utilizar un recurso se representa como una llamada `entry`, que debe ser aceptada por el correspondiente administrador del recurso. Se puede construir fácilmente con tareas de Ada un sistema estructurado con monitores y procesos, y viceversa; la elección entre un sistema u otro depende en gran medida del gusto personal. Probablemente, el esquema de Ada refleja más fácilmente el comportamiento de un sistema concurrente en una estructura distribuida, en la cual recursos ajenos son administrados por procesos que actúan como guardianes del recurso.

5.2.4.5 Características de Implementación

En un sistema concurrente, los procesos ora están suspendidos por alguna condición de sincronización, ora están potencialmente activos, es decir, no hay obstáculos lógicos para su ejecución. En general, sólo un subconjunto de los procesos potencialmente activos podrá ejecutarse, a menos que haya tantos procesadores como procesos potencialmente activos. En el caso de un sistema monoprocesador, sólo podrá ejecutarse un proceso a la vez. Entonces, es conveniente saber en cuál de los siguientes estados puede encontrarse un proceso (véase Figura 5.5):

- * Espera.
- * Preparado (Es decir, potencialmente activo pero sin ejecutarse por ahora).
- * Ejecución.

En programación concurrente, el programador no tiene un control directo sobre la velocidad de ejecución de los procesos. De hecho, el usuario no interviene en el cambio de estado de un proceso, por ejemplo, de preparado a ejecución (operación de selección en la Figura 5.5), ya que dicho cambio de estado lo realiza el sistema operativo. En la Figura 5.5 vemos que un proceso puede dejar el estado de ejecución y pasar al estado de preparado, como consecuencia de la operación expulsión.

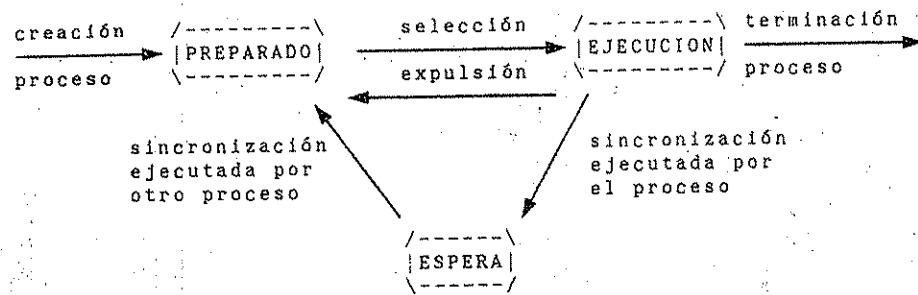


Figura 5.5 Diagrama de estados de un proceso.

La expulsión es una acción realizada por el sistema operativo, el cual obliga a un proceso a abandonar su estado de ejecución, incluso si, desde el punto de vista lógico, pudiera continuar ejecutándose felizmente. Se puede expulsar un proceso bien después de ejecutar una sentencia de sincronización, la cual hace que un proceso suspendido pase al estado preparado (p. ej., una operación `V` sobre un semáforo), bien cuando ocurre alguna otra condición, como puede ser que se le acabe el tiempo asignado (rodaja de tiempo). Después de expulsar un proceso, uno de los

procesos en estado preparado puede pasar al estado de ejecución. Este tipo de implementación permite al programador ver el sistema como un conjunto de actividades que se realizan en paralelo, aunque todas las ejecute el mismo procesador. El procesador sólo ejecuta un proceso cada vez, pero cada proceso sólo se ejecuta durante un tiempo limitado, una vez que reciba el control de otro proceso. Obsérvese, sin embargo, que es posible tener implementaciones de concurrencia no expulsora. En este caso, sólo cambia la ejecución a otro proceso cuando el proceso en curso se suspende deliberadamente a sí mismo o cuando solicita un recurso que no está disponible.

La parte del sistema operativo que soporta en tiempo de ejecución la transición de estados de la Figura 5.5 se denomina núcleo. Para ilustrar las características básicas de un núcleo, considérese el caso de un único procesador compartido por un conjunto de procesos. Para simplificar, omitimos los problemas de sincronización en los accesos a dispositivos de E/S y nos centramos en la interacción interna entre los procesos. Estudios más completos de estas características se tratan en los textos de sistemas operativos. Aquí sólo presentamos un visión de los problemas básicos que son relevantes para comprender las características de concurrencia de los lenguajes de programación.

Existe un descriptor de proceso por cada proceso, en el cual se guarda la información que precisa el núcleo. Dicha información se necesita para restaurar el proceso desde un estado de bloqueo o espera, al estado de ejecución. Esta información (denominada status del proceso) incluye (si se usan prioridades) la prioridad del proceso y toda la información necesaria para el procesador; tal como la identidad del proceso, punto de ejecución, contenido de los registros (contador de programa, acumulador, registro índice) etc. Una de las funciones del núcleo es salvar el status del proceso cuando queda suspendido y restaurar el status cuando el proceso pasa a ejecución.

Podemos imaginarnos el núcleo como un tipo abstracto de datos: oculta algunas estructuras de datos privadas y dispone de procedimientos que proporcionan la única forma de utilizar esas estructuras. Esas estructuras del núcleo están organizadas como colas de descriptores de procesos. Los descriptores de los procesos preparados se guardan en la COLA_DE_PREPARADOS. Hay también una COLA_DE_CONDICION para cada condición que puede suspender un proceso: es decir, hay una cola para cada semáforo y una para cada objeto declarado de tipo cola en un monitor. Cada una de las colas se utiliza para almacenar los descriptores de todos los procesos suspendidos en un semáforo o retenidos en la cola por el monitor. La variable EJECUCION contiene el descriptor del proceso en ejecución. En la Figura 5.6 se muestra una visión clásica de las estructuras de datos del núcleo.

El reparto de rodajas de tiempo se implementa con una interrupción de reloj. Cada interrupción activa a la siguiente operación del núcleo, el cual lleva a la COLA_DE_PREPARADOS el proceso que se está ejecutando y pasa un proceso del estado

preparado al estado de ejecución.

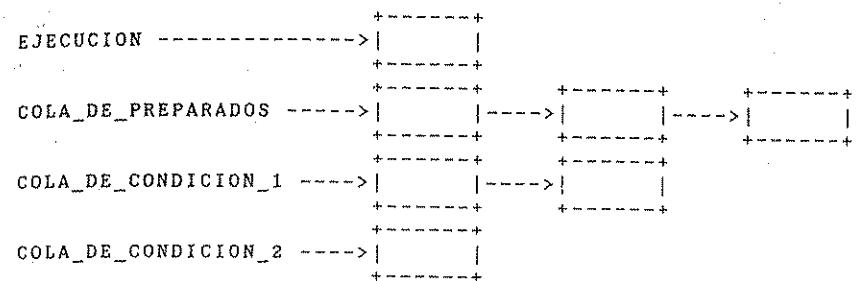


Figura 5.6 Estructura de datos del núcleo.

Operación Suspensión-y-Selección

1. Salvar en EJECUCION el status del proceso en ejecución.
2. Llevar EJECUCION a la COLA_DE_PREPARADOS.
3. Llevar un descriptor desde la COLA_DE_PREPARADOS a EJECUCION.
4. Restaurar el contenido de los registros con los valores que hay en EJECUCION.

5.2.4.5.1 Semáforos

Si el lenguaje dispone de semáforos, las primitivas P y V se pueden implementar como llamadas a procedimientos del núcleo. Una suspensión por la condición c causada por una operación P, se implementa en el núcleo de la siguiente forma:

Operación de Suspensión-según-Condición

1. Salvar en EJECUCION el status del proceso en ejecución.
2. Llevar EJECUCION a la COLA_DE_CONDICION_C.
3. Llevar un descriptor desde la COLA_DE_PREPARADOS a EJECUCION.
4. Restaurar el contenido de los registros con los valores que hay en EJECUCION. (Este paso sirve para activar el proceso).

La reactivación de un proceso que está en espera de la condición c, causada por una operación V, se implementa en el núcleo con la operación:

Operación de Reactivación

1. Salvar en EJECUCION el status del proceso en ejecución.
2. Llevar EJECUCION a la COLA_DE_PREPARADOS.
3. Llevar un descriptor desde la COLA_DE_CONDICION_c a la COLA_DE_PREPARADOS.
4. Llevar un descriptor desde la COLA_DE_PREPARADOS a EJECUCION.
5. Restaurar el contenido de los registros con los valores que hay en EJECUCION.

Con el fin de garantizar la indivisibilidad de las primitivas P y V, las operaciones correspondientes del núcleo se ejecutan sin interrupción de reloj. Esto no es necesario cuando la interrupción de reloj es la que invoca la operación suspensión-y-selección, ya que se puede asumir que la operación termina en una fracción de la rodaja de tiempo asignada, y por tanto es intrínsecamente indivisible. Sin embargo, si sería necesario incluso en este caso, y también en general, si se utilizan interrupciones de E/S para activar procesos que esperan operaciones de E/S.

5.2.4.5.2 Monitores

En el caso de los monitores, una forma sencilla de implementar la exclusión mutua consiste en inhabilitar las interrupciones de reloj cuando se llama a un procedimiento monitor y habilitarlas cuando se retorna del mismo. (En el ejercicio 5.16 se propone una implementación alternativa). Damos por supuesto que se pueden habilitar e inhabilitar con una única instrucción máquina, y además, que un registro especial indica si las interrupciones se encuentran habilitadas o no. Ese registro forma parte del status del proceso y debe salvarse en el descriptor del mismo cuando dicho proceso sea suspendido. Para mayor sencillez, asumimos también que los procedimientos monitores no contienen llamadas a otros procedimientos monitores. Cuando un proceso llama a un proceso monitor, se salva la dirección de retorno en una entrada del descriptor del proceso. Las operaciones delay y continue se pueden implementar con procedimientos del núcleo. De hecho, delay se implementa con la operación suspensión-según-condición, mientras que continue se implementa de la forma siguiente:

Operación Continuar

1. Salvar en EJECUCION el status del proceso en ejecución.
2. Habilitar las interrupciones en EJECUCION.

3. Llevar la dirección de retorno del monitor a la imagen del contador del programa que hay en EJECUCION.
4. Llevar EJECUCION a la COLA_DE_PREPARADOS.
5. Sea COLA_DE_CONDICION_c la cola citada por la sentencia continue. Si dicha cola no está vacía, llevar un descriptor de esa cola a EJECUCION; en caso contrario, llevar un descriptor a EJECUCION desde la COLA_DE_PREPARADOS.
6. Restaurar el contenido de los registros con los valores que hay en EJECUCION.

Obsérvese que se han utilizado deliberadamente algunos pasos de las operaciones del núcleo (p.ej., el paso 3 de suspensión-y-selección y el paso 3 de suspensión-según-condición) para resaltar la indeterminación. Estos pasos son exactamente los puntos en los cuales la implementación debe escoger la política oportuna para hacer una elección determinista. La parte del núcleo responsable de esta política se denomina planificador. Unas veces el planificador es muy simple (p.ej., todas las colas se gestionan como si fueran FIFO) y otras sofisticado (p.ej., teniendo en cuenta prioridades, tiempos de espera, etc).

5.2.4.5.3 Rendezvous

Veamos ahora algunas características de implementación del rendezvous de Ada. Hay una cola de tareas preparadas (COLA_DE_PREPARADOS). Cada entrada de cada tarea tiene un descriptor con los siguientes campos:

- * A Un valor lógico A que indica si la entrada está abierta (A=verdadero indica que la tarea está preparada para aceptar una llamada a esa entrada).
- * E Una referencia E a la cola de descriptores de tareas cuyas llamadas a la entrada están pendientes (cola_de_espera).
- * T Una referencia T al descriptor de la tarea a la cual pertenece la entrada.
- * I Una referencia I a la primera instrucción del cuerpo de la sentencia accept (para simplificar, asumimos que en una sentencia select no pueden aparecer dos sentencias accept para la misma entrada). Esta referencia es significativa sólo si la entrada está preparada para aceptar la llamada (A=verdadero).

Para mayor sencillez, asumimos también que la implementación de las sentencias de sincronización se hace con operaciones ininterrumpibles del núcleo, es decir, el núcleo inhabilita y habilita las interrupciones antes de ejecutar cada sentencia. Por

la misma razón, ignoramos el problema de cómo pasar parámetros entre tareas.

Veamos la implementación de la llamada a una entrada realizada con operaciones del núcleo:

1. Salvar en EJECUCION el status de la tarea en ejecución.
2. Llevar EJECUCION a la cola asociada con la entrada.
3. Si la entrada está abierta, entonces cerrar ($A=falso$) todas las entradas abiertas de la tarea llamada y llevar su descriptor a EJECUCION; actualizar el valor del contador de programa de EJECUCION con el valor almacenado en el campo I del descriptor. En caso contrario, llevar a EJECUCION un descriptor de la COLA_DE_PREPARADOS.
4. Restaurar el contenido de los registros con los valores que hay en EJECUCION.

Cuando se llega al end de una sentencia accept, el rendezvous se completa con las siguientes acciones del núcleo:

1. Salvar en EJECUCION el status de la tarea en ejecución.
2. Llevar a la COLA_DE_PREPARADOS el descriptor de la tarea llamante referenciado en el campo T del descriptor de la entrada.
3. Llevar EJECUCION a la COLA_DE_PREPARADOS.
4. Llevar a EJECUCION un descriptor de la COLA_DE_PREPARADOS.
5. Restaurar el contenido de los registros generales con los valores que hay en EJECUCION.

Las acciones a ejecutar como consecuencia de una sentencia accept no incluida en una select son:

1. Si la cola de espera de la entrada está vacía, entonces
 - 1.1 Poner a verdadero el campo A del descriptor de la entrada y salvar en I la dirección de la primera instrucción del cuerpo de la sentencia accept.
 - 1.2 Salvar en el descriptor de la tarea el status de la tarea en ejecución y poner en el campo T la referencia al descriptor de la tarea.
 - 1.3 Llevar a EJECUCION un descriptor desde la COLA_DE_PREPARADOS.

- 1.4 Restaurar el contenido de los registros generales con los valores que hay en EJECUCION.

2. Si la cola de espera no está vacía, entonces se acepta inmediatamente el rendezvous, es decir, continua la ejecución de la tarea hasta que se llega al final del cuerpo de la sentencia accept.

Para ejecutar una sentencia select, primero se construye una lista de las entradas abiertas que están implicadas en la selección. Si esta lista está vacía, entonces se produce la excepción SELECT_ERROR. En caso contrario, se precisan las siguientes acciones del núcleo:

1. Si una o más entradas de la lista tienen una cola no vacía, entonces
 - 1.1 Escoger una arbitrariamente.
 - 1.2 Aceptar inmediatamente el rendezvous, es decir, continuar la ejecución del cuerpo del accept de la alternativa seleccionada.
2. Si todas las colas de entradas están vacías, entonces
 - 2.1 Poner en estado "abierto" todas las entradas y salvar en el campo I del descriptor correspondiente la primera instrucción del accept de cada alternativa abierta.
 - 2.2 Salvar en el descriptor de la tarea el status de la tarea en ejecución y poner en el campo T la referencia al descriptor de la tarea.
 - 2.3 Llevar un descriptor desde la COLA_DE_PREPARADOS a EJECUCION.
 - 2.4 Restaurar el contenido de los registros generales con los valores que hay en EJECUCION.

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

Las estructuras de control a nivel de sentencia se empezaron a investigar con intensidad en los años 70. (Dijkstra 1968a) escribió el primer artículo insistiendo en la necesidad de una disciplina en la programación, y en la influencia de la sentencia goto en la producción de programas con poca claridad. Muchas investigaciones posteriores sobre programación estructurada estaban orientadas a describir estructuras de control idóneas para fomentar la escritura de programas bien organizados y legibles. (Bohm y Jacopini 1966) justifican formalmente los esfuerzos realizados con la presentación de un subconjunto limitado de estructuras de control (secuencia, if then else, do while) suficientes para poder escribir cualquier programa. (Knuth

1974) da una visión de las estructuras de control a nivel de sentencia y una comparación a fondo de muchos proyectos de lenguajes. (Wirth 1968) y (Van der Poel y Maersen 1974) describen ejemplos de extensiones para incorporar estructuras de control de alto nivel a lenguajes de nivel máquina.

(Liskov y otros 1977) detallaron las estructuras de control de CLU a nivel de sentencia definidas por el usuario. A su vez, (Atkinson y otros 1978) presentaron un modelo de implementación.

El paso de parámetros lo estudian (Gries 1971), (Pratt 1975) y (Organik y otros 1978). Este último utiliza el modelo descriptivo propuesto por (Johnston 1971) para analizar el problema. (Francez 1977) estudia el método de paso de parámetros por nombre.

El manejo de excepciones es la característica lingüística que describe cómo controlar programas y manejar sucesos imprevistos con el fin de mejorar la fiabilidad. (Randell 1975) y (Parnas y Wurges 1976) estudian la fiabilidad de los programas y su tolerancia a los fallos.

El manejo de excepciones en sí lo estudian (Goodenough 1975) y (Levin 1977). Las excepciones en PL/I las analiza (McLaren 1977). En CLU son analizadas por (Liskov y Snyder 1979) junto con una comparación con otros proyectos de lenguajes. La implementación del manejo de excepciones de CLU la describen (Atkinson y otros 1978).

La primera descripción de corrutinas llegó con (Conway 1963). Posteriormente, (Knuth 1973) y (Wegner 1968) dan una visión introductoria a las corrutinas. (Marlin 1980) preparó un texto sobre corrutinas que contiene un estudio de lenguajes, una detallada descripción semántica y un análisis de los métodos de programación.

La concurrencia en los lenguajes de programación se estudia normalmente como una rama de los sistemas operativos. El lector puede dirigirse a (Brinch Hansen 1973 y 1977) y (Haberman 1976) para estudiar la teoría de los sistemas operativos. El concepto de semáforo lo introdujo (Dijkstra 1968b y 1968c). El de monitor, (Brinch Hansen 1973) y (C. Hoare 1974). Los monitores se incorporan a los lenguajes Pascal Concurrente (Brinch Hansen 1975 y 1977), Modula (Wirth 1976b) y CSP/K (Holt y otros 1978a). Las nuevas primitivas de sincronización presentadas por (Hoare 1978) y (Brinch Hansen 1979) han tenido una gran influencia sobre el concepto de rendezvous de Ada. Dichas propuestas están dirigidas a proporcionar primitivas de sincronización para procesos y sistemas distribuidos. Los sistemas distribuidos están compuestos de una gran número de procesadores autónomos conectados entre sí por un medio de comunicación. Los sistemas distribuidos, al igual que las redes locales de pequeños ordenadores (p.ej., microprocesadores), tendrán probablemente una gran importancia en el futuro, por lo que se destinan a ese fin muchos medios para su investigación.

Ejercicios

- 5.1 Usted quiere diseñar una parte de un programa que lee una secuencia de valores enteros. La longitud de la secuencia no se conoce; pero acaba con un valor especial (p.ej., cero). Todos los valores leídos, excepto el final, deben procesarse de una manera indeterminada. La solución está en el siguiente fragmento en pseudopascal:

```
read unidad;
while unidad <> valor_final do
begin
  procesar unidad;
  leer otra unidad
end
```

En PLZ, el algoritmo quedaría

```
do leer unidad;
if unidad=valor_especial then exit;
procesar unidad
od
```

Comparar brevemente las dos soluciones en función de la legibilidad y facilidad de escritura.

- 5.2 En ALGOL 68 no hay bucle repeat...until.

- Describir cómo se puede simular dicho bucle. (Sugerencia: Se puede escribir un bucle sin cuerpo; la cláusula while puede ser una secuencia de sentencias que proporcionan un valor lógico).
- Analizar la solución obtenida bajo el punto de vista de legibilidad y fiabilidad de escritura.

- 5.3 Reescribir en ALGOL 68, Pascal y Ada el programa de PLZ de la Sección 5.1.3. Comentar la legibilidad y facilidad de escritura del programa en cada uno de los lenguajes.

- 5.4 En 1974, Zahn propuso dos nuevas estructuras de control "según suceso". Veamos ambas formas, con pequeños cambios sintácticos:

```
(A) begin quit on suceso_1, suceso_2, ...suceso_n;
    bloque_de_sentencias_0
  then
    suceso_1: bloque_de_sentencias_1;
    suceso_2: bloque_de_sentencias_2;
    ...
    suceso_n: bloque_de_sentencias_n
  end;
```

```
(B) do quit on suceso_1, suceso_2, ..., suceso_n;
    bloque_de_sentencias_0
  then
    suceso_1: bloque_de_sentencias_1;
    suceso_2: bloque_de_sentencias_2;
    ...
    suceso_n: bloque_de_sentencias_n
  od
```

El caso (A) especifica la ejecución de bloque_de_sentencias_0, mientras que el caso (B) indica la repetición de bloque_de_sentencias_0.

Hay una nueva sentencia, raise nombre_de_suceso, que indica que se ha producido un suceso. Esta sentencia puede aparecer en bloque_de_sentencias_0 de los casos (A) y (B), según el valor de nombre_de_suceso declarado en la cláusula quit on de la sentencia. El efecto de la sentencia raise nombre_de_suceso es una trasferencia de control a la sentencia siguiente a nombre_de_suceso en la cláusula then.

1. Reescribir el programa del Ejercicio 5.1 y el programa PLZ de la Sección 5.1.3, utilizando el bucle de Zahn.
 2. Expresar las estructuras if then else y case utilizando el tipo (A) de Zahn.
 3. Expresar las estructuras do while y repeat until de Pascal utilizando el tipo (B) de Zahn.
- 5.5 Ejecutar manualmente el programa tipo ALGOL para los siguientes casos: (a) llamada por referencia; (b) llamada por valor; (c) llamada por resultado; (d) llamada por valor-resultado; (e) llamada por nombre. Comparar los resultados obtenidos para cada caso en las variables a e i.

```
program paso_de_parametros;
  var i: integer;
  a: array[1..3] of integer;
  procedure mensaje(v: integer);
  begin
    v := v+1;
    a[i] := 5;
    i := 3;
    v := v+1
  end mensaje;
  for i := 1 to 3 do a[i] := 0;
  a[2] := 10;
  i := 2;
  call mensaje(a[i])
end paso_de_parametros
```

- 5.6 Se supone que la rutina siguiente intercambia el contenido de dos variables internas:

```
subprogram cambio(a,b: integer);
  var auxiliar: integer;
begin
  auxiliar := a;
  a := b;
  b := auxiliar
end cambio
```

Si a y b son parámetros por nombre, la rutina cambio no es conmutativa (p.ej., el efecto de cambio(c,d) es diferente al de cambio(d,c)).

- a.) Comprobar este hecho en el caso de cambio(i,a[i]), siendo a una matriz de enteros.
 - b.) ¿Qué ocurre si a y/o b se pasan por valor?
 - c.) ¿Qué ocurre si a y/o b se pasan por valor-resultado?
- 5.7 Exponer cómo se pueden implementar los efectos de los iteradores de CLU en un lenguaje que soporta corrutinas pero no iteradores.
- 5.8 Describir cómo se pueden utilizar los subprogramas "sensibles a la historia" de FORTRAN para simular corrutinas.
- 5.9 Una forma de manejar excepciones en un lenguaje de programación que no dispone de manejador de excepciones específico es devolver un código especial indicando la excepción alcanzada (Sección 2.3.2). Otra forma es enviar como parámetro el procedimiento de tratamiento de excepción (Sección 5.2.1.2). ¿Qué diferencia hay entre ambos métodos?
- 5.10 Los implementadores del lenguaje CLU citan la siguiente implementación como alternativa a la descrita en la Sección 5.2.2.1. En vez de tener una sola tabla de manejadores para un procedimiento, se construye una subtabla para cada llamada. Cada entrada en la subtabla corresponde con una excepción que puede alcanzarse en el procedimiento llamado.
1. Explicar cómo se puede construir la subtabla (en tiempo de compilación y para cada llamada). ¿Se puede hacer lo mismo para un esquema de manejo de excepciones del tipo PL/I?
 2. Indicar cómo se puede implementar la sentencia signal.
 3. Comparar el método de subtabla con el de tabla de manejadores en términos de rapidez y espacio.

4. Los criterios de implementación de los mecanismos de manejo de excepciones que han fijado las implementaciones de CLU son:

- a) La eficiencia de ejecución del caso normal no deberá verse afectada en absoluto.
- b) Las excepciones se han de ejecutar rápidamente, aunque no necesariamente al máximo de rapidez.
- c) Debe hacerse un uso razonable del espacio.

De acuerdo con estos criterios, ¿por qué es preferible el método de tabla frente al de subtabla?

5.11 Considérese un subprograma Ada que termina por error después de provocar una excepción. ¿Por qué puede producir diferentes resultados la implementación con paso de parámetros por copia de la de parámetros por referencia?

5.12 En el ejemplo de la Sección 5.2.4.1 del productor/consumidor implementado con semáforos, supóngase que escribimos P(exclumutua) en vez de V(exclumutua) en el proceso productor. ¿Cómo se comporta el sistema?

5.13 Cuando se utilizan semáforos para implementar la exclusión mutua, es posible asociar un semáforo SR con cada recurso R. Entonces, cada acceso a R se puede escribir como

```
P(SR);
    acceso(R);
V(SR)
```

¿Cómo se inicializa SR?

5.14 El ejemplo del productor/consumidor de la Sección 5.2.4.1 utiliza un semáforo para imponer una exclusión mutua sobre el acceso a la zona de memoria. ¿Se puede realizar una implementación de las operaciones poner y quitar de forma que no necesiten exclusión mutua? (Sugerencia: intente obtener una solución que no use la variable total).

5.15 Algunos ordenadores (p.ej., IBM 360) disponen de una instrucción indivisible, TS (test and set, comprobar y poner) que puede utilizarse para sincronización.

Sean X e Y dos variables de tipo lógico. La ejecución de la instrucción

```
TS(X,Y)
```

lleva el valor de Y a X y pone Y a falso.

Un conjunto de procesos concurrentes que han de ejecutar un conjunto de instrucciones con exclusión mutua,

pueden utilizar una variable lógica global PERMISO, inicializada a verdadero, y una variable lógica local X de la forma siguiente:

```
repeat TS(X,PERMISO)
until X;
instrucciones a ejecutar con exclusión mutua;
PERMISO := verdadero
```

1. En este caso, los procesos no se interbloquean, siempre están preparados (a esto se le llama espera ocupada). Comparar esta solución con la basada en semáforos, en la que P y V están implementadas por el núcleo.

2. Describir cómo implementar las primitivas P y V mediante semáforos, utilizando las primitivas comprobar y poner, en un esquema de tipo espera ocupada.

5.16 Hemos implementado la exclusión mutua de los procedimientos monitores inhabilitando las interrupciones. Otra solución sería utilizar un semáforo para cada monitor y ejecutar una primitiva P sobre el semáforo antes de entrar al procedimiento monitor, y una primitiva V al salir. Especifique esta implementación y compare ambas soluciones.

5.17 Explicar cómo se puede utilizar una tarea Ada para implementar un semáforo.

CORRECCION

"?...curar programas enfermos o ... producir programas sanos?" (Wegner 1979)

Ya hemos visto en los Capítulos 1 y 2 que la corrección y la fiabilidad se han convertido en objetivos prominentes de la producción de software. Al principio de la era de los ordenadores estos objetivos eran simplemente obvios e implícitos. Hoy, ese punto de vista se considera bastante ingenuo. Ahora se reconoce que si se espera alcanzar tales objetivos, se debe tener especial cuidado y tomar medidas esmeradas. Al centrarse en las consideraciones de corrección, se ha afectado conjuntamente a los métodos y a los lenguajes de programación. El efecto sobre estos lenguajes también se ha hecho sentir indirectamente a causa de los nuevos métodos. Este capítulo describe el desarrollo de las ideas de corrección en los lenguajes de programación.

6.1 CORRECCION Y FIABILIDAD

Nuestra primera tarea es definir con precisión los términos que hemos venido usando informalmente hasta ahora. Un programa es correcto si cumple sus especificaciones. Por otra parte, un programa es fiable si es muy probable que cuando pedimos un servicio al sistema, lo ejecute a nuestro gusto. Como tal, la fiabilidad es difícil de cuantificar, ya que está relacionada con la calidad del sistema tal y como el usuario la percibe. Es natural que pesen más ciertas situaciones en las que se necesita con urgencia un servicio del sistema, que aquellas en las que se le pide algo rutinario. Es más, a menudo, a un sistema se le considera fiable incluso aunque no satisfaga la noción estricta de corrección. Esto puede ser debido a que:

1. El error no afecta al uso del sistema, y es fácilmente detectado y corregido por el usuario. Por ejemplo, un error de deletreo en un mensaje tal como:

PRAMETRO N. 3 FUERA DEL MARGEN

no disminuye la fiabilidad del sistema. Por el contrario, señalar un número de parámetro incorrecto, sí podría hacer disminuir la fiabilidad.

2. El error no se da muy a menudo o en casos críticos. Por ejemplo, en un sistema de comutación de paquetes, en situaciones extremas, la (infrecuente) pérdida de un paquete de caracteres pertenecientes a un texto largo, recibido en una línea de entrada, puede ser tolerada. En cambio, la pérdida de datos sería inaceptable si los datos fueran usados para manejar una planta nuclear en tiempo real, en la que hay que tratar situaciones de emergencia tan pronto como se reciban datos críticos en el ordenador.

Por el contrario, un sistema cuya corrección haya sido

rigurosamente fijada, puede no ser fiable. De hecho, la corrección se define relativa a la especificación, de tal forma que un programa puede ser correcto incluso aunque no satisfaga las necesidades del usuario. Las razones básicas son:

1. Las especificaciones reflejan los requisitos equivocadamente o de forma incompleta. Por ejemplo, los requisitos pueden especificar que se necesita cierta autorización para poder enterarse de la cantidad de depósitos bancarios de cada consumidor, pero las especificaciones no indican cómo y cuándo se va a comprobar tal autorización.
2. Las especificaciones no establecen lo que se supone que el sistema debe hacer en casos "anómalos", tales como fallos en el soporte hardware/software, o errores en los datos de entrada. Por eso, el sistema se comporta correctamente si se cumplen varias suposiciones acerca del entorno, pero ciertos datos de entrada inesperados (y no detectados) podrían causar serias malfunciones del sistema, ya que no se ha previsto cómo tratar los casos excepcionales.

Aunque la fiabilidad es el principal objetivo, es difícil de cuantificar, y se usa a menudo la corrección como sustitutivo, con la esperanza de que las especificaciones captén adecuadamente las propiedades deseadas del sistema. Es más, la corrección puede establecerse en términos precisos, e incluso se han desarrollado métodos apropiados para probar la corrección de programas. El resto de este capítulo, tratará principalmente de la corrección.

El proceso de garantizar (certificar) la corrección de un programa se denomina certificación. Un programa se puede certificar de varias formas, incluyendo la prueba experimental del programa, la verificación del mismo, o una combinación de ambas. El deseo de facilitar la certificación de programas ha influido en el diseño de lenguajes, hasta el punto de que, por ejemplo, los lenguajes Euclid, Gypsy y Ada tienen la verificabilidad como un objetivo específico. Trataremos los efectos de estos objetivos en los lenguajes de programación.

Estos efectos se pueden ver a varios niveles. El efecto más global, y más importante, está en los mecanismos de estructuración de programas proporcionados por el lenguaje. Los programas grandes y complejos son objetos no manejables. El único modo de manejar esta complejidad es dividir el programa en partes pequeñas, de tal forma que cada una de ellas pueda ser certificada individualmente. La próxima tarea es certificar estas pequeñas partes combinadas para formar el conjunto deseado.

A otro nivel, lo concerniente a la corrección ha enfocado la atención en construcciones individuales utilizadas para escribir las partes pequeñas. A otro nivel más, se ha sometido a examen la combinación de estas construcciones.

En este capítulo examinaremos los dos últimos niveles. Las técnicas de estructuración de programas se examinarán en el próximo capítulo.

6.2 REVISIÓN DE PROGRAMAS

Después de escribir un programa, ¿cómo saben los programadores que este programa es el que ellos querían escribir? En otras palabras, ¿cómo se convencen a sí mismos, aparte de a otras personas, de que lo que han producido es un programa correcto? El proceso que siguen para hacer esto se conoce como "revisión de programas". Es más fácil mostrar la corrección de algunas soluciones y programas que de otros, es decir, hay algunos programas cuya lógica es más fácil de entender que la de otros. Más adelante tratamos informalmente algunas características de lenguajes que hacen difícil la revisión de los programas. En las Secciones 6.3 y 6.4 se estudiará un enfoque más formal.

No obstante, es importante darse cuenta de que la estimación de la corrección del software no deberá estar confinada a la fase de programación. En realidad, varios estudios experimentales muestran que el coste de la corrección de errores en la fase de codificación es mucho más alto que el corregirlos en la fase de diseño. Se han propuesto técnicas apropiadas, y ahora son comúnmente usadas en la práctica, para permitir a los directores de proyectos gestionar un proyecto a través de todas las fases de su desarrollo, de tal forma que los errores sean detectados lo antes posible.

La revisión de diseño es una de tales técnicas. En la fase de diseño, la persona responsable del diseño que se está revisando presenta documentos y describe oralmente las decisiones de diseño a un grupo de personas familiarizadas con dicho proyecto. De esta forma, se descubren fácilmente especificaciones vagas o malentendidos. A esta misma técnica, cuando se aplica en la fase de programación, se le denomina inspección de código. En este caso, uno o más programadores que no sean los autores del programa, revisan dicho programa con objeto de descubrir errores. A menudo, cuando uno de los programadores no consigue comprender algunos aspectos del programa y pide alguna explicación, el autor descubre la presencia de un error. Está claro que el lenguaje de programación puede ayudar en este proceso hasta el punto de hacer que la escritura de programas los haga fácilmente legibles.

Las características de los lenguajes que complican el seguimiento o revisión de programas, se tratan en la Sección 6.2.1. La Sección 6.2.2 analiza las construcciones introducidas por recientes lenguajes para facilitar el proceso de revisión.

6.2.1 Características Dañinas de los Lenguajes

La sentencia goto es el ejemplo más conocido de característica dañina de un lenguaje. En su famoso artículo acerca del efecto de la sentencia goto (Dijkstra 1968a), Dijkstra dijo que los programas que contienen muchos goto's también tienden a contener muchos errores. El argumento de Dijkstra va más allá de la mera proposición de la abolición de la sentencia goto. La razón por la que la sentencia goto es mala estriba en que dificulta la revisión del programa. Y lo hace rompiendo la continuidad secuencial de las sentencias del programa, y violando el requisito de que la estructura de la solución esté reflejada en la estructura del programa.

La sentencia goto, por supuesto, no es la única sentencia dañina. Otras dos características de los lenguajes que dificultan la revisión de programas son los efectos laterales y el alias. Ambas permiten que una sentencia del programa tenga efectos que no son evidentes examinando la sentencia de forma aislada. Tal sentencia, obviamente, reduce la legibilidad del programa. Los efectos laterales y el alias se tratan por separado en las Secciones 6.2.1.1 y 6.2.1.2.

6.2.1.1 Efectos Laterales

En la Sección 3.4 se definieron los efectos laterales como modificaciones de un entorno no local. Los efectos laterales se utilizan principalmente para proporcionar un método de comunicación entre unidades de programa. Esta comunicación se puede establecer por medio de variables globales. Sin embargo, si el conjunto de variables globales usado para este propósito es grande, y cada unidad tiene un acceso sin restricciones a él, el programa se vuelve difícil de leer y de comprender. Cada unidad puede, potencialmente, referenciar y modificar toda variable del entorno global, quizás de un modo diferente de aquel para el que estaba destinada la variable. El problema consiste en que una vez que una variable global se usa para comunicaciones, es difícil distinguir entre efectos laterales deseables e indeseables. Por ejemplo, si la unidad u1 llama a la unidad u2, y u2 inadvertidamente modifica una variable global x usada para comunicación entre las unidades u3 y u4, una llamada a u2 produce un efecto lateral no deseado. Tales errores son difíciles de encontrar y corregir, ya que el síntoma no se asocia fácilmente a la causa del error (Nótese que un simple error tipográfico podría llevarnos a este mismo problema). Otra dificultad está en que el solo examen de la instrucción de llamada no revela las variables que pueden ser afectadas por la llamada. Esto reduce la legibilidad de los programas, ya que, en general, se debe inspeccionar el programa completo para comprender el efecto de una llamada.

La comunicación por medio de accesos no restringidos a variables globales es particularmente peligrosa cuando el programa es grande y está compuesto de varias unidades que han

sido desarrolladas independientemente por varios programadores. Una forma de reducir estas dificultades es encaminar la comunicación entre unidades por medio de parámetros. Sin embargo, la sobrecarga causada por el paso de parámetros, podría hacer inaceptable esta solución en aplicaciones con tiempos críticos. Por otra parte, debe ser posible restringir el conjunto de variables globales mantenidas en común por dos unidades, a las estrictamente necesarias para la comunicación entre ellas. También puede ser útil especificar que una unidad puede leer pero no modificar ciertas variables. Estos y otros problemas serán ampliamente tratados en el próximo capítulo.

Los efectos laterales también se dan en el paso de parámetros por referencia, donde se usa un efecto lateral para modificar el parámetro real. El programador debe tener cuidado para no producir efectos laterales no deseados en los parámetros reales modificando los parámetros formales correspondientes. El mismo problema surge con las llamadas por nombre.

La consecuencia de los efectos laterales puede ser especialmente molesta en los subprogramas de tipo función. A estos subprogramas se les llama escribiendo el nombre del subprograma en una expresión, tal como en

$w := x + f(x, y) + z$

En presencia de efectos laterales, en Pascal por ejemplo, la llamada a f podría producir un cambio en x o y (si se pasan por referencia), o incluso en z (si z es global a la función) como consecuencia de un efecto lateral. Esto reduce la legibilidad del programa, ya que no se puede confiar en la conmutatividad de la adición en general. En el ejemplo, si f modifica x por un efecto lateral, el valor producido para w es diferente si x es evaluada antes o después de la llamada a f.

Además de afectar a la legibilidad, los efectos laterales pueden impedir que el compilador optimice el código utilizado para la evaluación de ciertas expresiones. En el ejemplo

$u := x + z + f(x, y) + f(x, y) + x + z$

La función f y la subexpresión $x + z$ no se pueden evaluar una sola vez.

6.2.1.2 Alias

Dos variables son alias si denotan (comparten) el mismo objeto durante la activación de una unidad (ver Sección 3.4). La modificación de un objeto bajo un nombre de variable, es automáticamente visible a todas las variables que comparten el objeto. Se puede ver un ejemplo en la sentencia EQUIVALENCE de FORTRAN. Por ejemplo, las sentencias

EQUIVALENCE (A,B)

A = 5.4

asocian el mismo objeto a A y B, y ponen su valor a 5.4. Como consecuencia, las sentencias

B = 5.7

WRITE (6,10)A

imprimirían 5.7, a pesar de que el valor explicitamente asignando a A sea 5.4. La asignación a B afecta a la vez a A y a B.

El alias puede establecerse durante la ejecución de un procedimiento cuando los parámetros se pasan por referencia. Considerese el siguiente procedimiento Pascal, del que se supone que intercambia los valores de dos variables enteras sin usar ninguna variable local.

```
procedure intercambia (var x,y: integer);
begin
  x := x+y;
  y := x-y;
  x := x-y
end
```

Antes de la ejecución, se le propone al lector que compruebe el procedimiento, intentando comprender en qué circunstancia opera apropiadamente.

La respuesta es "generalmente sí"; de hecho, el procedimiento trabaja apropiadamente excepto cuando los dos parámetros reales son la misma variable, como en la llamada

intercambia (a,a)

En este caso, el procedimiento pone a a cero, ya que x e y se convierten en alias, y así, cualquier asignación a x y a y en el mismo procedimiento, afecta a la misma posición de memoria. El mismo problema puede derivarse de la llamada

intercambia (b[i], b[j])

cuando los índices i y j tienen el mismo valor.

Los punteros también pueden crear problemas. Por ejemplo, la llamada

intercambia (p[], q[])

no intercambia los valores apuntados por p y q, si p y q apuntan al mismo objeto.

El alias anteriormente indicado se da en cualquiera de las dos condiciones siguientes:

1. Los parámetros formales y los reales comparten los mismos objetos.

2. Las llamadas a procedimientos tienen solapamiento en los parámetros reales.

El alias también puede darse cuando un parámetro formal (por referencia) y una variable global denotan los mismos (o solapados) objetos. Por ejemplo, si el procedimiento intercambia se vuelve a escribir como

```
procedure intercambia (var x: integer):
begin
  x := x+a;
  a := x-a;
  x := x-a
end
```

siendo a una variable global, la llamada

intercambia (a)

genera un resultado incorrecto, a causa del alias entre x y a. Es interesante hacer notar que el alias no ocurre si los parámetros se pasan por copia; tales parámetros actúan como variables locales del procedimiento, y los parámetros reales correspondientes sólo se ven afectados a la salida del procedimiento.

Los punteros son intrínsecamente generadores de alias, tal y como muestra el siguiente fragmento de un programa Pascal.

```
var p, q : ^T;
      .
      .
new(p);
q := p
```

Los punteros p y q apuntan al mismo objeto. De esta forma, es posible cambiar el valor de p[] con una asignación a q[], y viceversa. ALGOL 68 puede generar también alias apuntando a objetos con nombre en la pila. Esta posibilidad, como ya hemos visto, ha sido erradicada en Pascal y otros lenguajes.

Los inconvenientes del alias afectan a los programadores, lectores e implementadores. Los subprogramas son difíciles de comprender, ya que ocasionalmente puede denominarse el mismo objeto mediante diferentes nombres. Este fenómeno no se puede descubrir inspeccionando el subprograma, sino que se requiere el examen de todas las unidades que puedan llamar a tal subprograma. Como consecuencia del alias, un subprograma puede producir resultados incorrectos e inesperados. El alias también disminuye la posibilidad de generación de código optimizado. Por ejemplo, en el caso

```
a := (x-y*z)+w;
b := (x-y*z)+u;
```

la subexpresión $x-y*z$ no puede ser evaluada una sola vez y utilizada en las dos asignaciones, si a es un alias de x , y , o z .

6.2.2 Características de los Lenguajes Disciplinados

El objetivo de la producción de programas correctos, ha influido en el diseño de varios lenguajes de programación modernos, los cuales restringen o eliminan por completo la utilización de características dañinas. En la Sección 5.1.4 se analiza el caso de la sentencia goto. Algunos lenguajes modernos no ofrecen la sentencia goto (p.ej., Bliss, Euclid, Gypsy). Otros la han mantenido como un mecanismo para sintetizar estructuras de control legítimas que no estén suministradas por el lenguaje. De manera similar, varios lenguajes actuales restringen o eliminan las fuentes originarias de efectos laterales.

Euclid y Gypsy son dos ejemplos de lenguajes diseñados con el expreso objetivo de facilitar la escritura de programas correctos. Muchas de las características mencionadas en la Sección 6.2.1 se han erradicado, siendo la mayoría gotos, alias y efectos laterales en funciones. Gypsy tampoco proporciona variables globales. Euclid permite variables globales, pero provee mecanismos para controlar su acceso. Euclid y Gypsy proporcionan buenos casos para estudiar los efectos de la eliminación de características dañinas.

Como muchos lenguajes de finales de los años 70, Euclid y Gypsy están basados en Pascal. El principal objetivo de estos dos lenguajes es permitir la escritura de programas verificables. Las características dañinas mencionadas anteriormente, se vieron como características particularmente problemáticas para la verificación del programa y, como hemos visto, para la revisión de programas, por lo cual no se han incluido en ninguno de los dos lenguajes. Euclid y Gypsy son los lenguajes más conocidos que se basan en tal enfoque. Para simplificar, restringiremos nuestra discusión a Euclid.

El alias se definió anteriormente como la habilidad para acceder, con más de un nombre, al mismo objeto en la misma unidad de activación. Básicamente, hay dos caminos para eliminar el alias. Uno consiste en acabar por completo con las características que hacen posible el alias, como punteros, parámetros por referencia, datos globales y matrices. Sin embargo, esto nos llevaría a un lenguaje muy pobre. El otro camino, tomado por Euclid, consiste en poner restricciones al uso de tales características para excluir la posibilidad del alias.

En el caso de parámetros por referencia, el problema sólo surge si los parámetros reales se solapan. Si dichos parámetros reales son variables simples, es necesario asegurarse de que todos ellos son distintos. Así, la llamada

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

`p (a,a)`

se considera ilegal en Euclid. El paso de una matriz y uno de sus componentes, también está prohibido. Por ejemplo, la llamada

`p (b[1],b)`

a un procedimiento cuya cabecera sea

`procedure p (var x: integer; var y: array [1..10] of integer)`

es ilegal ya que $y[1]$ y x serían alias. Las formas anteriores de alias ilegal, se pueden detectar en tiempo de compilación. Sin embargo, la llamada

`intercambia (b[i], b[j])`

al procedimiento `intercambia`, descrito en la Sección 6.2.1.2, genera alias sólo si i es igual a j . Se podría sencillamente prohibir tales llamadas, pero el resultado sería un lenguaje torpe y difícil de usar. Euclid especifica que en tal caso, el compilador genere la condición

`i ≠ j`

como un aserto de legalidad. En la fase de pruebas, los asertos de legalidad, se pueden compilar automáticamente como comprobaciones en tiempo de ejecución, usando cierta opción del compilador. Si en tiempo de ejecución un aserto se evalúa como falso, la ejecución se aborta, produciendo un mensaje apropiado. Sin embargo, el principal uso de los asertos de legalidad está en la verificación del programa. De hecho, el sistema Euclid incluye un verificador de programa, y un programa se considera correcto solamente cuando el verificador comprueba que todos los asertos son verdaderos.

El manejo del alias en presencia de punteros es más complejo. Considerar de nuevo el fragmento del programa Pascal tratado en la Sección 6.2.1.2.

`var p,q : [T];`

`;`

`new (p);`

`q := p;`

El problema del alias entre p y q se trata del mismo modo que el problema de las matrices y los elementos de matrices, es decir, p y q se ven como selectores que hacen referencia a un componente de una colección de objetos del tipo T , del mismo modo que $b[i]$ y $b[j]$ hacen referencia a un componente de la matriz b . Una asignación a $b[i]$ o a $b[j]$ se ve como una asignación al objeto b completo, lo cual sucede al cambiar un valor almacenado en una porción del mismo. Así, $b[i]$ y $b[j]$ no se ven como alias.

ya que nombran explícitamente el mismo objeto b. De igual manera, una asignación a p_j o a q_j se puede considerar como una modificación de la colección de los componentes de tipo T; así, p_j y q_j no son alias, ya que nombran explícitamente a la misma colección.

Este podría parecer un punto de vista ingenioso pero engañoso sobre el alias con punteros. En realidad, diferentes estructuras de datos pueden estar compuestas de componentes del mismo tipo T, generados dinámicamente. Ver una asignación a p_j como una asignación a la colección, es decir, como una modificación de cualquiera de tales estructuras de datos, no sirve realmente de ninguna ayuda.

Para permitir un nivel adicional de comprobación de punteros que no producen solapamiento, se introduce un nuevo mecanismo, la colección (Ver Sección 4.5.3). Se requiere que el programador divida todos los objetos dinámicos en colecciones separadas, e indique qué punteros pueden apuntar a qué colecciones. Cada puntero solamente puede estar ligado a una colección. Una asignación entre dos punteros es legal sólo si ambos punteros apuntan a la misma colección.

La detección de un alias ilegal entre dos punteros, causado por llamadas a procedimientos, es similar al caso de las matrices. De hecho, una colección C y un puntero ligado a C, son similares a una matriz y a una variable usada como índice. La referenciación es exactamente igual que la indexación en una matriz. Por ejemplo, si p y q apuntan a la misma colección, y p_j y q_j se pasan ambos como parámetros, para que no haya solapamiento es necesario que se compruebe que

$p \neq q$

como aserto de legalidad.

Como ya vimos en la sección anterior, el alias también se puede dar entre variables globales y parámetros formales de un procedimiento. En Euclid, la detección de alias en tales casos no requiere un trabajo adicional. De hecho, las variables globales deben ser explícitamente importadas por el subprograma que las necesite, y deben ser accesibles en todo el ámbito en el que el subprograma es llamado. Para cada variable importada, es también necesario indicar si va a ser leída, escrita o ambas cosas. Así, las variables globales modificables pueden ser tratadas por el algoritmo de detección de alias como parámetros adicionales implícitos pasados por referencia.

La importación explícita de variables globales permite al programador restringir el conjunto de variables visibles en un procedimiento a cualquier subconjunto de variables (no enmascaradas) declaradas en ámbitos más externos. El compilador puede así asegurar que en una determinada unidad sólo se accede a las variables visibles, y que cualquier acceso es legal. Por ejemplo, puede comprobar que no se intenta modificar una variable

que sólo es de lectura. Esto es una ventaja sobre las reglas de ámbito de tipo ALGOL, especialmente para programas grandes, en los que los procedimientos internos heredan automáticamente todas las variables (no enmascaradas) declaradas en los bloques que los encierran, y pueden modificarlas de forma incontrolada. Este punto se tratará más ampliamente en el siguiente capítulo.

Finalmente, no se permite que las funciones de Euclid tengan parámetros por referencia, y solamente pueden importar variables de lectura. De esta manera, su ejecución no puede causar efectos laterales, comportándose realmente como funciones matemáticas.

Una consecuencia importante de la prohibición del alias en los procedimientos, es que el paso de parámetros por referencia sea igual al paso de parámetros por copia (ver Sección 5.2.1). Por lo tanto, la elección de cómo realizar el paso de parámetros la puede hacer el compilador basado exclusivamente en razones de eficiencia. El lector recordará que Ada no especifica cuándo el paso de parámetros se realiza por referencia o por copia. Al contrario que Euclid, Ada no requiere que el alias ilegal de un programa sea detectado antes del tiempo de ejecución por un verificador de programa. Así, algunos programas Ada ilegales pueden permanecer sin ser detectados, y como consecuencia, una implementación diferente del paso de parámetros puede producir diferentes resultados para el mismo programa.

El enfoque de Euclid (y Gypsy) es ciertamente interesante, y las soluciones adoptadas son claras. Algunas restricciones impuestas por Euclid no pueden ser tratadas por un compilador tradicional, y requieren un sistema de desarrollo de programas que incluya un verificador. En concreto, todos los asertos de legalidad necesitan ser probados por el verificador. El éxito de este enfoque tomado por Euclid y Gypsy depende en gran medida de la aceptación que la verificación del programa encuentre entre los programadores. Por el momento es un tema controvertido, debido, en parte, a la falta de experiencia en verificación de aplicaciones en tiempo real.

6.3 COMPROBACION DEL PROGRAMA

Tratando de revisar los programas, una de las herramientas a nuestra disposición es el ordenador. A la actividad de usar un ordenador para ayudar a detectar errores de programa, se la conoce como comprobación del programa. La prueba del programa es un tipo de comprobación. La comprobación de sintaxis y de tipos que realiza un compilador son otros tipos de comprobación de programa. De hecho, hay un espectro de comprobaciones que se pueden hacer a un programa, desde las puramente estáticas, realizadas por un simple compilador, a las completamente dinámicas, realizadas por un probador de programas. Esta sección examina estos tipos de comprobaciones y sus implicaciones en el diseño de lenguajes.

6.3.1 Comprobaciones Estáticas

Las comprobaciones estáticas son las que se pueden realizar sin ejecutar el programa. Ya hemos visto (en el Capítulo 4) que los lenguajes con gran rigidez de tipos permiten que la corrección de tipos se compruebe en tiempo de compilación. En la última sección vimos que la especificación de variables globales importadas por un procedimiento permite otro nivel más de comprobación estática. En otro nivel más, podemos comprobar cosas tales como el posible uso de una variable antes de una asignación, dos asignaciones consecutivas a una variable, o la asignación a una variable que no se usa nunca. Aunque puede que estas acciones no produzcan errores en una ejecución particular del programa, son síntomaticos de posibles problemas. Tales errores se pueden detectar estáticamente según un modelo del flujo de control del programa. Ya que tal modelo no es necesario para la traducción del programa a código máquina, los compiladores tradicionales no lo construyen y por lo tanto no son capaces de detectar las condiciones anómalas citadas. El modelo del flujo de control, normalmente un grafo, es necesario, ya que las preguntas que formulamos son de la forma "¿Existe un camino de flujo de control para el cual cierta condición es verdadera?". A tal comprobación se la conoce como análisis del flujo de datos.

6.3.2 Comprobaciones en Tiempo de Ejecución

Hay condiciones que se pueden formular estáticamente pero no pueden ser evaluadas de forma estática. Tales comprobaciones deben posponerse hasta la fase de ejecución o deben ser evaluadas por un verificador de programas. Así, en Euclid, el requisito para que dos parámetros por referencia reales no produzcan solapamiento, requiere que para la llamada $p(a[i], a[j])$, se cumpla la condición $i \neq j$. Aunque la condición puede ser construida por el compilador, se pasa al verificador, o bien en fase de prueba, se puede transformar en una condición que debe ser comprobada en tiempo de ejecución. Otro ejemplo es la comprobación en tiempo de ejecución para asegurar que los índices de las matrices están dentro de los márgenes especificados.

6.3.3 Prueba

La prueba, como ya hemos dicho, es otro tipo de comprobación del programa. Requiere la ejecución del programa con un conjunto de datos de entrada, enfocado a poner de manifiesto algunos posibles errores. A menudo, también se produce una impresión (traza) que muestra, en algunos puntos del programa, los valores de ciertas variables seleccionadas para ayudar a descubrir la causa de un error. Cualquier que haya intentado probar un programa, conoce la dificultad de esta actividad y la carencia de cualquier criterio cuantitativo para la medida del éxito de la prueba. No obstante, existen algunos criterios para seleccionar un conjunto de datos que puedan garantizar un cierto grado de

seriedad en la prueba. Por ejemplo, los datos de la prueba deberán ser de tal forma que (1) todas y cada una de las sentencias se ejecuten al menos una vez, o (2) todas y cada una de las ramas del flujo de control se ejecutarán al menos una vez, o (3) todos y cada uno de los caminos de control se ejecuten al menos una vez (si el número de caminos en el programa es infinito, como por ejemplo, en presencia de bucles, se selecciona un conjunto finito).

Una exposición detallada de la prueba de programas se sale fuera del ámbito de este libro; el lector interesado en el detalle, puede consultar la literatura especializada referenciada en la sección de Ampliación Bibliográfica al final del capítulo. No obstante, nos gustaría hacer énfasis en que ningún criterio práctico de prueba puede garantizar la ausencia de errores. En concreto, es fácil mostrar que los tres criterios mencionados anteriormente son inadecuados incluso para un tipo de programas muy restringido.

Supongámos que nuestros programas requieren solamente una secuencia de sentencias de asignación lineales, es decir, sentencias de la forma

$$y := a_1*x_1 + a_2*x_2 + \dots + a_m*x_m + b$$

donde y , x_1 , x_2 , ... x_m son variables del programa, y a_1 , a_2 , ..., a_m , b son constantes. Un programa P con variables de entrada x_1 , x_2 , ... x_m y variables de salida y_1 , y_2 , ... y_n (se supone que estas variables son disjuntas) puede ser considerado como puntos de la representación de una función de un espacio de m dimensiones en puntos de un espacio de n dimensiones. La función que representa cada m -tupla (x_1, x_2, \dots, x_m) en un valor de y_i , siendo $1 \leq i \leq n$, puede ser considerada como un hiperplano en un espacio de $(m+1)$ dimensiones. Cualquiera de tales planos, para $1 \leq i \leq n$, puede ser determinado de manera única dando $m+1$ puntos (linealmente independientes). De esta forma, podemos concluir que la prueba del programa puede garantizar la corrección si los hiperplanos identificados por nuestra prueba son coincidentes con los hiperplanos deseados. Se necesitarían al menos $m+1$ conjuntos de datos de prueba linealmente independientes para probar el programa exhaustivamente, mientras que bastaría una ejecución de prueba para satisfacer cada uno de los tres criterios ya mencionados.

6.3.4 Ejecución Símbólica

Otra forma más de comprobación de programas es la ejecución símbólica, la cual trata de superar el problema de los casos en los que el número de pruebas es excesivamente pequeño, ejecutando el programa con datos simbólicos y deduciendo los valores de los datos de salida como funciones de los datos de entrada. En el curso de la ejecución símbólica del programa, también es posible detectar algunas anomalías de éste, tales como caminos por los que nunca se pasa (lo cual implica código

inútil). La ejecución simbólica no ha tenido realmente éxito en la práctica, y el análisis de las causas cae fuera del propósito de este texto. El lector interesado puede consultar el material referenciado en la sección de Ampliación Bibliográfica.

6.4 VERIFICACION DEL PROGRAMA

En la verificación de un programa, se intenta demostrar matemáticamente que el programa es correcto, es decir, que cumple sus especificaciones. Si después de probar un programa, sólo se puede estar seguro de que el programa se ejecuta correctamente para los datos particulares que se habían elegido, con la verificación se intenta garantizar que el programa se ejecuta correctamente para cualquier dato legal de entrada. Este objetivo es, por supuesto, más ambicioso y mucho más arduo.

6.4.1 Semánticas Formales

El efecto más importante del trabajo sobre verificación ha sido la motivación de definiciones precisas de la semántica de los lenguajes de programación. Si se espera probar que un programa concreto cumple una tarea específica, lo primero que necesitamos saber es lo que hace cada sentencia del programa. En el Capítulo 3 se mostró la variante operacional de la definición semántica, la cual describe la semántica de un lenguaje de programación por medio de un modelo abstracto de la ejecución de los programas escritos en tal lenguaje. El modelo describe la creación y manipulación de las estructuras básicas de datos que necesita el procesador para la ejecución de programas. El método es estimable desde el punto de vista pedagógico, ya que sugiere un modelo abstracto para la implementación. Sin embargo, el nivel de la descripción operacional expuesta en el Capítulo 3 es demasiado informal para utilizarlo en demostraciones matemáticas.

Se han propuesto varios métodos formales. Por ejemplo, el método axiomático, que será brevemente ilustrado aquí, el cual está basado en lógica matemática. Según este método, un estado de un cálculo no se describe mediante el contenido de las estructuras de datos del procesador, sino por una expresión lógica de las variables del programa (llamada predicado o aserto) que debe ser cierta en ese estado.

Se llama postcondición de S a un predicado P que debe cumplirse después de una sentencia S. A un predicado Q tal que la ejecución de S termina y la postcondición P se cumple a su término, se le conoce como precondición de S y P. Por ejemplo, $y=3$ es una precondición para la sentencia $x := y+1$ (siendo x e y variables enteras) y la postcondición $x > 0$. El predicado $y \geq 0$ es también una precondición para la sentencia $x := y+1$, y la postcondición es $x > 0$. Realmente, $y \geq 0$ es la mínima precondición, es decir, la precondición necesaria y suficiente para que la sentencia $x := y+1$ lleve a la postcondición $x > 0$. Se dice que un

predicado W es la mínima precondición para la sentencia S y una postcondición P, si cualquier precondición Q para S y P implica W es decir, W se cumple para cualquier otra precondición Q. La implicación se denota como " \rightarrow ". En el ejemplo, tenemos que

$$y=3 \rightarrow y \geq 0$$

En general, dada una sentencia de asignación $x := E$ y una postcondición P, la mínima precondición se obtiene reemplazando cada aparición de x en P por la expresión E. La mínima precondición se escribe como $P_x \rightarrow E$. En el ejemplo, $P_x \rightarrow y+1$ es $y+1 > 0$, es decir, $y \geq 0$.

Normalmente, la semántica de un lenguaje de programación se puede describir por una función wp (llamada transformador de predicados) que para cualquier sentencia S y cualquier postcondición P, tiene como su propio valor la mínima precondición. W. Esto se escribe como

$$wp(S, P) = W$$

En el caso de la sentencia de asignación $x := E$ tenemos que

$$wp(x := E, P) = P_x \rightarrow E$$

Esta caracterización de la asignación es correcta siempre que la evaluación de la parte derecha de la sentencia no tenga un efecto lateral. Es más, la variable asignada no puede ser un alias de otras variables del programa. En tales casos, la ejecución de la sentencia puede también afectar a variables que no aparecen en la parte izquierda de la sentencia, y el transformador de predicados dado sería incorrecto. Esto es un ejemplo de cómo los efectos laterales y el alias pueden complicar la revisión, tanto formal como informal, de los programas.

Las sentencias simples, tales como sentencias de asignación, se pueden combinar en acciones más complejas por medio de estructuras de control de nivel de sentencia. Por lo tanto, se necesitan reglas de composición para caracterizar el efecto semántico de la combinación de sentencias individuales en un segmento del programa.

Por ejemplo, en el caso de la secuencia, si nosotros sabemos que

$$wp(S_1, P) = Q$$

y que

$$wp(S_2, Q) = R$$

entonces

$$wp(S_2; S_1, P) = R$$

Los casos de selección e iteración son más complejos. En el examen que hacemos a continuación usaremos esquemas de selección e iteración basados en los comandos de guarda tratados en el Capítulo 5.

Si P es la postcondición que se debe establecer mediante la sentencia if:

```
if B1 --> S1
  B2 --> S2
  .
  .
  Bn --> Sn
fi
```

la mínima precondition es un predicado que debe expresar los siguientes hechos:

- Al menos una guarda debe ser cierta, de otra forma, el programa se aborta y no se puede satisfacer P.
- Por cada una de las ramas que tengan una guarda cierta (*), la correspondiente lista de sentencias debe establecer la verdad de P.

Formalmente, esto se puede escribir como

```
wp (estado-if,P) = (B1 or B2 or . . . or Bn) and
(B1 --> wp (S1,P) and (B2 --> wp (S2,P) and . . .
and Bn --> wp (Sn,P))
```

Por ejemplo, dado el siguiente fragmento de programa (siendo x, y y max enteros)

```
if x>y --> max := x
  x<y --> max := y
fi
```

y la postcondición

```
(max = x and x>y) or (max = y and y>x)
```

se demuestra fácilmente que la mínima precondition es cierta, es decir, no se necesita imponer ninguna restricción en las variables para asegurar la postcondición deseada.

Supongamos ahora que P es la postcondición que se debe establecer mediante el siguiente bucle:

(*) Recuérdese que puede haber más de una, y cualquiera de ellas puede ser elegida de forma no determinista.

```
do B1 --> S1
  B2 --> S2
  .
  .
  Bn --> Sn
od
```

El problema aquí es que no sabemos cuántas veces se va a repetir el cuerpo del bucle. En efecto, si supiéramos, por ejemplo, que el número de iteraciones fuese n, la construcción anterior sería equivalente a la composición secuencial de n sentencias if, con el mismo conjunto de comandos de guarda, y así su semántica sería sencilla (basada en la semántica de la secuencia y de la sentencia if).

Para superar esta dificultad, abandonaremos el objetivo de obtener la mínima precondition y caracterizaremos la semántica de la sentencia do mostrando cómo una precondition suficiente Q se puede derivar de una postcondición dada P. Ya que nuestro objetivo es escribir programas correctos, la verdad de Q asegura la verdad de la mínima precondition (desconocida) y por tanto la verdad de P (la postcondición deseada).

La condición Q debe ser tal que

- El bucle termina.
- A la salida del bucle, P se cumple.

Así, el predicado Q se puede escribir como $Q = T \text{ and } R$, donde T implica la terminación del bucle y R implica la verdad de P a la salida de él.

La creación de estos dos predicados T y R que satisfacen estas dos propiedades no es sencilla. Por razones de simplicidad, ignoraremos el problema de la terminación del bucle y enfocaremos nuestra atención en la creación de R. Supongamos que podemos identificar un predicado I que se cumple antes y después de cada iteración del bucle y, a la salida de éste, (por ejemplo, cuando ningún Bi es verdad) I implica P. A I se le conoce como predicado invariante para el bucle. Formalmente, I satisface las siguientes condiciones:

- I and (B1 or . . . or Bn) --> wp (IF,R)
 donde IF es la sentencia obtenida al reemplazar las palabras clave do y od del bucle por if y fi.
- I and not (B1 or B2 or . . . or Bn) --> P

Si podemos identificar un predicado I que satisface (i) y (ii), entonces podemos tomar a I como el predicado deseado R, ya que P se cumple a la terminación si se cumple que $R = I$ antes de ejecutar el bucle.

En la próxima sección se da un ejemplo del uso de

invariantes.

6.4.2 Verificación de programas

No es la intención de este libro el estudio en profundidad de la teoría de la verificación de programas y de las herramientas necesarias para la verificación automática de programas. Sin embargo, ahora estamos en situación de establecer con precisión cómo se puede demostrar que los programas son correctos, y de echar una ojeada a sus implicaciones. En primer lugar, los requisitos de corrección del programa se deben especificar formalmente dando dos predicados: una precondition IN (o aserto de entrada) en las variables de entrada, y una postcondición OUT (o aserto de salida) sobre las variables de entrada y las de salida. El trabajo de la verificación consiste en mostrar que si IN se cumple antes de ejecutar el programa, la ejecución termina en un estado en el que se cumple OUT. Esta prueba requiere el uso de una caracterización semántica de sentencias tal como se describió en la sección anterior.

La verificación de programas se ilustra aquí con la ayuda de un ejemplo simple. Considerar el siguiente fragmento de programa, en el cual se supone que todas las variables son enteros.

```
i := k;
sum := k;
do i > 1 --> i := i-1;
    sum := sum+i
od
```

Sea el aserto de entrada

IN: $k > 0$

y el aserto de salida

```
    k
OUT: sum =  $\sum_{j=1}^k j$  and  $k > 0$ 
```

Queremos probar que este fragmento de programa es correcto respecto a IN y OUT.

Obviamente, la terminación del bucle está asegurada, ya que la variable i sólo se altera por la instrucción $i := i-1$, y por lo tanto solamente puede tener valores en una secuencia descendente, con lo que al final $i > 1$ se hará falsa.

Se puede demostrar fácilmente que, como invariante, el predicado

```
    k
I: sum =  $\sum_{j=1}^k j$  and  $0 < i \leq k$ 
```

satisface las condiciones (i) y (ii) de la sección anterior. Así, al empezar la ejecución del bucle con valores de variables que satisfacen I, se asegura la terminación en un estado en el que se satisface P.

Por último, es fácil probar que I se cumple después de las instrucciones

```
i := k;
sum := k;
```

si la precondition $k > 0$ se cumple antes de las dos instrucciones.

En conclusión, si IN se cumple antes de ejecutar el fragmento anterior, la ejecución termina en un estado en el que OUT se cumple, es decir, el programa es correcto con respecto a IN y OUT.

Dados los asertos de entrada y de salida, la verificación del programa comienza derivando asertos intermedios en varios puntos del programa que se debe demostrar que se cumplen. En particular, como ya hemos visto, los asertos intermedios los debe suministrar el verificador (hombre o máquina) en forma de invariantes de bucle. Otros ejemplos de asertos intermedios son los asertos de legalidad generados por el compilador de Euclid y los asertos que se pueden especificar explícitamente por el programador en Euclid, por medio de la sentencia assert. El propósito de la sentencia assert de Euclid es suministrar, como parte integrante del programa, asertos intermedios que deben ser probados por el verificador. De esta manera, los asertos se convierten en parte de la documentación del programa. Es más, hay opciones que permiten al programador transformar sentencias assert en comprobaciones automáticas en tiempo de compilación, durante la fase de pruebas, o alternativamente suprimir su evaluación. Por último, si se cuenta con un verificador en el conjunto de herramientas del sistema de programación, los asertos se pueden probar por medio del verificador, con lo que el programa se certifica completamente de una forma estática.

La influencia de las características de los lenguajes de programación en el proceso de revisión se deja sentir tanto en los lectores humanos como en los verificadores automáticos. Muchas de las características que hacen difícil la revisión de los programas para los humanos, también son difíciles de tratar por un verificador de programas. Por ejemplo, los efectos laterales en las funciones complican la evaluación de la mínima precondition para las asignaciones, tal como muestran los siguientes ejemplos. Sea $y := f(x)+z$ una sentencia de asignación, y $P(z)$ un predicado de la variable z que se debe cumplir como postcondición de la asignación. La ausencia de efectos laterales garantiza que $P(z)$ es también la mínima precondition. Sin embargo, la posibilidad de efectos laterales requiere el examen de la función f , ya que z podría ser modificada por un efecto lateral. Es más, el verificador, bien sea humano o automático, debe tener mucho cuidado si el alias está permitido por el

lenguaje. En el ejemplo, $P(z)$ no sería la mínima precondición si $z = y$ fuesen alias.

La verificación de programas está todavía en su infancia, y es tema de debate el cuándo se convertirá en una práctica del ordenador. No obstante, la verificación de programas ha tenido una profunda influencia en la programación y en sus lenguajes. Estimula un enfoque riguroso de la programación y proporciona una definición formal de los lenguajes. Incluso en ausencia de un verificador de programas, la revisión rigurosa de la corrección del programa puede ayudar al programador a descubrir posibles errores. Los asertos intermedios que deberá demostrar el verificador (por ejemplo, los invariantes de bucle) se pueden considerar como comprobaciones en tiempo de ejecución, y proporcionan un instrumento muy útil para la certificación de programas por medio de pruebas sistemáticas.

Antes de terminar este capítulo, queremos explicar por qué hemos tratado la prueba y la verificación a pesar de que nuestro objetivo principal sean los lenguajes de programación. Básicamente hay dos razones. La primera es que estas áreas cada vez tienen más influencia en el diseño de los lenguajes. La segunda, y más importante, es que ya que el motivo de los lenguajes de programación es el desarrollo de software, el estudio de técnicas para construir software fiable es parte integral del estudio de los lenguajes. Aunque el estudio en profundidad de estos temas no es el objetivo de este libro, este capítulo intenta familiarizar al lector con los principios básicos, y proporciona un punto de partida para un estudio más amplio.

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

Corrección de programa y fiabilidad son dos aspectos importantes de la ingeniería del software que han emergido como temas populares desde el principio de los años 70. El artículo de Parnas en (Yeh 1977a) versa sobre "fiabilidad frente a corrección", y muestra cómo la estructura del programa puede tener influencia en la fiabilidad. (Boehm 1976) muestra cómo la pronta detección de errores y la corrección en los programas puede reducir los costes de producción. (Yourdon 1975) y (Baker 1972) describen técnicas prácticas tales como la revisión de diseño y la inspección de código.

En varios artículos se analizan las construcciones de los lenguajes de programación que hacen penoso el análisis de los programas. A la famosa carta de Dijkstra sobre la sentencia goto (Dijkstra 1968a), le siguieron otros artículos sobre otros aspectos de los lenguajes. (Hoare 1975a y b) y (Kieburz 1976) analizan el papel de los punteros. (Wulf y Shaw 1973) trata el papel de las variables globales. (Popek y otros 1977) comenta los efectos laterales y el alias. (Reynolds 1979) clasifica la mayoría de estas características dañinas. Estos hallazgos han tenido una fuerte influencia en el diseño de varios lenguajes

recientes; tales como CLU, Alphard, Gypsy, Euclid y Rusell.

La comprobación estática del programa se trata parcialmente en varias partes de este libro (por ejemplo, el Capítulo 4 trata extensamente la comprobación de tipos). El papel del análisis del flujo de datos para detectar anomalías en el programa se estudia en (Fosdik y Osterweil 1976) y (Hecht 1977). Se dan otros argumentos en (Williams 1979).

(Myers 1979) proporciona una visión amplia de las actividades implicadas en las pruebas de software. Enfoques formales de las pruebas de programas se tratan en varios artículos agrupados en (Yeh 1977b) y examinados por Goodenough en (Wegner 1979). El estudio sobre la poca adecuación de los criterios de pruebas tratado en la Sección 6.3.3 está tomado de (Tai 1980).

La ejecución simbólica se presenta en (King 1976). Los sistemas de desarrollo que incorporan un ejecutor simbólico los describen (Cheatham y otros 1979) y (Asirelli y otros 1979).

Los fundamentos matemáticos de la verificación de programas fueron tratados por (Floyd 1967) y (Hoare 1969). Una presentación detallada de la teoría de Floyd y Hoare se describe en (Manna 1973). El estado del arte en este campo se puede encontrar en (Yeh 1977b) y en el artículo de London en (Wegner 1979). (DeMillo y otros 1979) arguye que la verificación del programa no se puede usar en la práctica para garantizar la fiabilidad del software, a causa de la naturaleza de las pruebas. Cita ejemplos de matemáticos en cuyas pruebas de los teoremas se han encontrado errores años después de que su veracidad había sido aceptada. Nuestra presentación del método axiomático está basada en (Dijkstra 1976). No obstante, es importante observar que el énfasis de Dijkstra no está en la verificación del programa, sino en una método para el desarrollo de programas correctos. Primero, desarrolla un lenguaje simple que hace posible la escritura de programas elegantes y da una definición formal para cada construcción del lenguaje en términos de transformadores de predicados. Segundo, muestra un cálculo tal que, dados los predicados de entrada y salida, se permite derivar un programa que es correcto respecto a los predicados. Por esto, el enfoque de Dijkstra es constructivo; no se prueba la corrección del programa después de haberlo escrito, sino que se deduce que es correcto por el cálculo. Dicho enfoque se presenta con el desarrollo de varios ejemplos de poca a moderada complejidad, y no está claro si se puede (y cómo se puede) usar en aplicaciones más grandes y más complejas. No obstante, el libro representa una contribución fundamental al desarrollo de una disciplina de programación, y su lectura es altamente recomendable.

Ejercicios

- 6.1 Suponga que se le dan las siguientes especificaciones de un programa.

El programa evalúa e imprime el factorial de un entero positivo cuyo valor se lee como un dato de entrada.

?Es correcto el programa Pascal listado a continuación? ¿es fiable?

```
program factorial (input,output);
var i, n, fact:integer;
begin read (n);
  fact := n; i := n;
  while i<>1 do
    begin i := i-1;
    fact := fact*i
    end;
  write (fact)
end.
```

- 6.2 Considerar la siguiente función Pascal:

```
function f(x,y:integer):integer;
  function g(z:integer):integer;
  begin
    .
    .
    end;
  begin
    f := g(x)+g(y)
  end.
```

1. Dar argumentos convincentes (aunque informales) sobre la verdad o falsedad de la sentencia "Para cualquier cuerpo de la función g, las llamadas $f(a,b)$ y $f(b,a)$ producen los mismos resultados para cualquier par de variables enteras a y b".

2. Cambiemos el modo del paso de parámetros de las dos funciones f y g a una llamada por referencia, y supongamos que a f siempre se la llama como $f(a,a)$, siendo a una variable entera. Comentar cuándo podemos transformar con seguridad la sentencia $f := g(x)+g(y)$ en $f := 2*g(x)$.

- 6.3 Poner ejemplos simples que muestren la diferencia en el número de ejecuciones entre los criterios de pruebas 1 y 2 mencionados en la Sección 6.3.3.

- 6.4 Poner ejemplos sencillos de programas sin bucles que muestren la diferencia en el número de ejecuciones entre los criterios de pruebas 2 y 3 mencionados en la Sección 6.3.3.

- 6.5. Suponga que está tratando de definir (manualmente) datos de prueba que satisfagan uno de los criterios citados en la Sección 6.3.3. Comentar brevemente el efecto de las sentencias goto.

- 6.6 Suponga que modificamos las sentencias if con guarda de Dijkstra, tal como se muestra a continuación:

```
if B1 --> S1
  B2 --> S2
  .
  .
  BN --> SN
else SE
fi
```

La rama else se selecciona sólo si ninguna rama de guarda tiene una guarda true. Dar el transformador de predicado para esta sentencia.

- 6.7 Describa la semántica de las siguientes sentencias Pascal en términos de transformadores de predicado:

- Sentencia if (sin parte else).
- if then else.
- Bucle for.

- 6.8. Dado el aserto de entrada

IN: $n > 0$

y el aserto de salida

OUT: $\text{fact} = n*(n-1)*(n-2)* \dots * 1$

probar formalmente la corrección del programa del Ejercicio 6.1 con respecto a IN y OUT.

**PROGRAMACION
A GRAN
ESCALA**



"Falibilidad humana - de grande a grandiosa"
(Mills 1979)

La producción de grandes programas (aquellos que están formados por varios miles de líneas) presenta una serie de problemas que no se dan en el desarrollo de los pequeños. Los mismos métodos y técnicas que funcionan bien con los programas pequeños, no son necesariamente aplicables a los grandes. Una vez que se ha reconocido que tales métodos no se pueden aplicar, se puede enfocar la producción de grandes programas siguiendo tres caminos distintos.

- 1.- Técnicas de dirección que proporcionan un control sobre las personas asignadas a un proyecto, que permiten seguir los progresos realizados.
- 2.- Métodos de diseño que permiten abordar los problemas que se dan en los programas grandes. Estos métodos han influido en el desarrollo de lenguajes que soportan facilidades para este tipo de programación.
- 3.- Herramientas de desarrollo que ayudan en las etapas creativas y automatizan las no creativas.

Para acentuar las diferencias que surgen entre la producción de programas pequeños y grandes, DeRemer y Kron han introducido los términos de "Programación a gran escala" y "Programación a pequeña escala". El punto 1 refleja la diferencia más relevante entre ambos tipos de programación. Si una persona puede producir todo el software necesario para un proyecto, todos los problemas girarán en torno a la experiencia de ese programador y a la disponibilidad de un entorno adecuado de programación. El desarrollo de sistemas de software a gran escala requiere varios programadores, y no existe un verdadero problema de programación, sino más bien de dirección y coordinación. Una dirección adecuada debe ser capaz de estimar los recursos suficientes para realizar una tarea, asignando al proyecto los recursos necesarios en el tiempo apropiado, dividiendo el esfuerzo del desarrollo en fases claras y bien definidas, controlando el desarrollo de cada fase mediante revisiones periódicas de diseño e imponiendo un serie de convenios para cada actividad. Aron dice que el "énfasis en la dirección en lugar de en la tecnología, representa uno de los grandes cambios en la naturaleza de la programación desde los años cincuenta" (Aron 1974). Aunque muchos de estos problemas se salen del objetivo de este libro, los lectores interesados pueden encontrar la literatura adecuada en la Sección de Sugerencias para Ampliación Bibliográfica.

Este capítulo trata a fondo los puntos 2 y 3 mencionados anteriormente, que por otra parte están muy relacionados. Por ejemplo, una biblioteca de programas puede verse como una herramienta o como un componente necesario en un método basado en el desarrollo incremental y en la producción de software reutilizable.

El capítulo está organizado de la siguiente forma: La Sección 7.1 indica la diferencia entre la programación a pequeña y gran escala; con la ayuda de un ejemplo. La Sección 7.2 revisa los métodos de diseño y marca las diferencias entre los métodos para programas grandes y para los pequeños. La Sección 7.3 evalúa las distintas características de los lenguajes actuales para soportar la programación a gran escala. La Sección 7.4 discute la necesidad de sistemas de desarrollo, en los cuales el lenguaje y un adecuado conjunto de herramientas se deben integrar para proporcionar un entorno adecuado para la programación.

7.1 ¿QUE ES UN PROGRAMA GRANDE?

El concepto de programa grande es difícil de definir. No queremos igualar el tamaño de un programa (número de sentencias fuente) con su complejidad. Decir que un programa es grande, se refiere más al "tamaño" y complejidad del problema que se está resolviendo, que al tamaño final del programa. No obstante, normalmente, el tamaño de un programa es una buena medida de la complejidad del problema que se va a resolver.

Considérese la tarea de construir un sistema de reservas para una compañía de líneas aéreas. El sistema deberá tener una base de datos con información de los vuelos. Los agentes de la aerolínea trabajando desde lugares remotos, pueden acceder a la base de datos en un tiempo y orden arbitrario. Pueden solicitar información de vuelos, así como horarios y precios; hacer o cancelar una reserva en un determinado vuelo; actualizar información existente, como el teléfono de un determinado pasajero. Determinadas personas autorizadas pueden acceder a la base de datos para operaciones especiales, como añadir o cambiar un vuelo, o cambiar el tipo de avión. Otros pueden acudir al sistema para obtener estadísticas sobre un vuelo en particular; o de todos los vuelos.

Un problema de esta magnitud impone severas restricciones en la estrategia a seguir para solucionarlo. Las características de tales problemas incluyen lo siguiente:

- El sistema tiene que funcionar correctamente. Un error aparentemente pequeño, tal como perder una lista de reservas o intercambiar dos listas diferentes, podría ser extremadamente costoso, y la corrección del sistema debe garantizarse a toda costa.
- El sistema ha de tener una larga vida. El coste asociado con un sistema de este tipo es alto, y no es práctico reemplazarlo totalmente por un nuevo sistema, ya que normalmente el coste del sistema sólo se podrá amortizar a largo plazo.
- Durante su periodo de vida, el sistema puede experimentar considerables modificaciones. Por ejemplo, puede aparecer una nueva regulación gubernamental que

requiera un cambio de la estructura de los precios, o que un nuevo tipo de avión tenga que ser añadido, etc. Otros cambios pueden ser beneficiosos, ya que la experiencia con el sistema puede haber puesto de manifiesto algunas debilidades de éste.

- Por la magnitud del problema, mucha gente (decenas o cientos de personas) están involucradas en el desarrollo del sistema.

Estas características imponen varios requisitos, tanto en el proceso del desarrollo del sistema, como en las herramientas utilizadas.

- El trabajo debe dividirse entre distintas personas. El trabajo asignado a cada persona debe ser claro y sin ambigüedades. Una persona, no tiene por qué conocer los detalles del trabajo de otra, aunque sus trabajos interactúen entre sí.

- El sistema se construye mediante partes desarrolladas independientemente por varias personas. A estas partes las llamaremos módulos. Estos módulos se deben diseñar y probar independientemente, lo cual tiene la ventaja de que algunos de ellos pueden aprovecharse de sistemas ya existentes. Análogamente, sería bueno que estos módulos pudieran volver a utilizarse en futuros proyectos, con lo que la aplicación entera podría volverse a utilizar, con pequeñas modificaciones, para distintas líneas aéreas.

- El sistema se debe de poder modificar fácilmente, es decir, ha de ser posible cambiar el interior de un módulo, sin afectar al sistema completo.

- Debe ser posible demostrar la corrección del sistema, basándose en la corrección de los módulos que lo constituyen.

Estas características nos muestran el concepto de un programa grande. Estos problemas son menos importantes cuando la programación es tarea de una sola persona. Por ejemplo, habría menos necesidad de dividir el trabajo, y por lo tanto no habría que poner tanto énfasis en las especificaciones de la interconexión entre módulos. Existe la posibilidad, al menos en teoría, de rehacer el sistema desde el principio, por lo que la modificabilidad (facilidad de modificación) es menos importante. El correcto funcionamiento únicamente implica a un módulo y por lo tanto es fácil de probar. En otras palabras, un sistema grande está compuesto por un conjunto de módulos que interactúan. Si no se adopta un método sistemático de diseño, cada módulo puede interactuar con otros de alguna forma sutil, con lo que cada módulo no podrá ser diseñado, entendido y probado, sin la intervención de los demás módulos. La complejidad de tales sistemas los haría imposibles de dirigir.

El límite entre la programación a grande y a pequeña escala, es difícil de establecer de una forma rigurosa. Sin embargo, podemos asumir que la programación a gran escala conduce al problema de la descomposición del sistema en módulos, mientras que la programación a pequeña escala se refiere a la producción de módulos individuales. Los métodos usados para estas dos actividades se tratan en la Sección 7.2. La Sección 7.3 evalúa los lenguajes de programación para clarificar la programación a gran escala.

7.2 ?PROGRAMACION A PEQUEÑA O A GRAN ESCALA? METODOS DE DISEÑO

La investigación en los métodos de diseño ha tomado el diseño "top-down" (descendente) como un buen camino para vencer las dificultades de la producción de software. En el diseño top-down, un problema se descompone iterativamente en subproblemas que se pueden resolver independientemente.

Cuando se aplica a programas pequeños, el método se denomina refinamiento por pasos sucesivos o paso a paso. Este método se describe como el proceso de escribir y reescribir un programa. En cada paso del desarrollo, el programa consiste en unas declaraciones y unas sentencias, algunas válidas para el lenguaje de programación, y otras abstractas que deberán refinarse en el paso siguiente. En cada paso, las declaraciones y sentencias abstractas se deben encontrar codificadas como comentarios, con el fin de obtener documentación. El proceso continúa hasta que al final todo el texto es un programa válido para el compilador.

Aunque la programación por refinamientos sucesivos es efectiva en programas de pequeña escala, falla en la programación a gran escala. Una de las razones es que no facilita las partes comunes a varios módulos. Los programadores, no están animados a representar partes comunes por una única abstracción, que ha de ser refinada una sola vez e invocada cada vez que se la necesita, sino que cada parte es refinada por separado. Otra razón es que el programa final no refleja adecuadamente el proceso de diseño, aunque las sentencias abstractas se encuentren como comentarios en el texto. Además las sentencias abstractas se escriben normalmente en una prosa informal, y suele ser necesario leer los refinamientos realizados para comprender qué es lo que hacen estas sentencias (*). Consecuentemente la legibilidad y modificabilidad de los programas, puede verse seriamente perjudicada en los programas de gran tamaño.

El diseño top-down de grandes aplicaciones, debe disponer de un sistema de descomposición en programas de pequeño tamaño (módulos). Siguiendo el principio del ocultamiento de la información, el diseñador debe distinguir claramente entre lo que

(*) Este último problema se puede resolver poniendo las sentencias abstractas en una notación formal.

hace el módulo (lo que exporta el módulo para el uso de otros módulos) y cómo está hecho el módulo (detalles internos.) La interconexión de un módulo debe especificar exactamente las entidades internas exportables para ser utilizadas por otros módulos, así como las externas importadas de otros módulos. El diseño de un módulo consiste en definir su interconexión, e iterativamente realizar una serie de módulos subsidiarios utilizados por este módulo.

Como ejemplo, considérese el problema de las reservas de una línea aérea que vimos en la Sección 7.1. Por ejemplo, podríamos tener un módulo de vuelo que proporcione información sobre los vuelos. Dado un número de vuelo, el módulo proporcionaría el número total de asientos, la hora de salida, las posibles conexiones con otros vuelos, etc. El módulo también proporcionaría operaciones de actualización, por ejemplo, la actualización de la hora de salida. También podríamos tener un módulo de reservas que maneje las listas de reservas para todos los vuelos. Dando un número de vuelo y una fecha, permitiría hacer o cancelar una reserva en un determinado vuelo. Un módulo de estadísticas proporcionaría operaciones estadísticas de uno o de todos los vuelos. El módulo de reservas deberá tener un acceso restringido al módulo de vuelo. Podrá pedir información sobre vuelos (por ejemplo, el número de asientos), pero no podrá realizar ninguna actualización. El módulo de reservas, se podría hacer de la siguiente forma.

```
module reserva
    import function asientos (numero_de_vuelo) return entero
    from vuelo;
    export procedure hacer_reserva(numero_vuelo,cliente);
    procedure cancelar_reserva(numero_vuelo,cliente);
```

La función asientos importada desde el módulo vuelo, recibe un parámetro del tipo numero_de_vuelo y devuelve un entero. Los procedimientos hacer_reserva y cancelar_reserva tienen parámetros del tipo numero_vuelo y cliente, y actualizan la lista de pasajeros para un determinado vuelo.

El diseño se completa cuando se han definido todas las interconexiones entre módulos. Sólo a este nivel podremos comenzar la implementación del cuerpo del módulo. La separación entre las fases de diseño e implementación distinguen este enfoque (descomposición en módulos) del refinamiento paso a paso. En el refinamiento paso a paso, el diseño y la codificación se realizan conjuntamente. Aquí, primero se descompone el sistema en módulos, y después se resuelve el problema de la implementación de los cuerpos de los módulos (utilizando para esto los refinamientos sucesivos).

El sistema de descomposición modular puede realizarse utilizando dos tipos concretos de abstracciones: abstracciones de procedimientos y abstracciones de datos. Las abstracciones de procedimientos, proporcionan un camino entre la entrada y la salida de datos. Las abstracciones de datos son un conjunto de

operaciones que manipulan un tipo concreto de datos. En cada etapa del diseño descendente, el problema a resolver se reduce al diseño de una iterativamente abstracción. Cada abstracción se descompone en abstracciones subsidiarias hasta que el programa está descompuesto en varias partes de complejidad moderada.

7.3 CARACTERISTICAS DEL LENGUAJE PARA LA PROGRAMACION A GRAN ESCALA

Esta sección trata del soporte que los lenguajes de programación proporcionan a la programación a gran escala; en concreto, del método de diseño que se presentó en la Sección 7.2. La mayoría de los lenguajes de programación proporcionan herramientas para descomponer los programas en pequeñas unidades autónomas. Estas unidades se pueden denominar módulos físicos. Utilizaremos el término módulo lógico para referirnos a un módulo en la etapa de diseño. Un módulo lógico puede estar formado por uno o más módulos físicos.

Organizaremos la discusión de acuerdo con los tres criterios siguientes:

1. ¿Qué es la noción de módulo físico soportado por el lenguaje, y cuán bien capta las propiedades lógicas especificadas en la etapa de diseño?
2. ¿Cómo se puede construir un programa combinando módulos físicos? ¿Cómo la estructura del programa impuesta por el lenguaje, refleja la descomposición modular jerárquica del diseño?
3. ¿Cómo se pueden implementar módulos físicos independientes? En concreto, ¿qué vida y posibilidad de volverse a usar tienen los módulos físicos?

La discusión se centrará en torno a Pascal, SIMULA 67 y Ada. Pascal puede considerarse como el representante de los lenguajes de tipo ALGOL. Las conclusiones sobre Pascal, salvo pequeños cambios, se pueden aplicar a otros miembros de la clase representada, tales como ALGOL 60 y ALGOL 68. Se harán algunos comentarios sobre GLU en la sección dedicada a Ada.

7.3.1 Pascal

7.3.1.1 Módulos

Las únicas aportaciones del Pascal a la descomposición modular son los procedimientos y las funciones. Sin embargo, los procedimientos y funciones sólo sirven para implementar la abstracción de procedimientos, mientras que para la implementación de las abstracciones de datos, debemos describir separadamente las declaraciones de tipos y procedimientos.

(funciones). En efecto, el lenguaje no proporciona ningún mecanismo para encapsular datos. El resultado de esto es que no existe una correspondencia biunívoca entre los módulos lógicos, que se identificaron durante la fase de diseño y los módulos físicos del programa.

7.3.1.2 Estructura de Programa

Todo programa Pascal tiene la siguiente estructura.

```
program nombredelprograma(ficheros);
  declaraciones de constantes,tipos,variables,
  procedimientos y funciones;

begin
  sentencias (sin declaraciones)
end.
```

Un programa está formado por declaraciones y operaciones. Las operaciones son las que proporciona el lenguaje o aquellas que se declaran como funciones o procedimientos. Una función o procedimiento puede contener en sí mismo la declaración de otras funciones y/o procedimientos. La organización del Pascal es por lo tanto, una estructura en árbol de módulos (árbol con anidamiento estático - Sección 3.6). La estructura en árbol representa un anidamiento textual de los módulos de nivel inferior. El anidamiento se utiliza para controlar el ámbito de actuación de las unidades declaradas dentro de los módulos, de acuerdo con la regla de ligadura estática que se presentó en la Sección 3.6.

Con el fin de evaluar la estructura de los programas en Pascal, consideremos el siguiente ejemplo. Supongamos que el diseño modular descendente de un módulo A, identifica dos módulos B y C que proporcionan abstracciones de procedimientos subsidiarios. Análogamente, el módulo B hace referencia a dos abstracciones de procedimientos privados, suministradas por los módulos D y E. El módulo C hace referencia a la abstracción de un procedimiento privado denominado F. La Figura 7.1 nos muestra una estructura anidada del programa que satisface las restricciones de diseño.

Un problema básico en la solución de la Figura 7.1, es que la estructura no refuerza las restricciones en las llamadas a procedimientos que se encontraron en la etapa de diseño. En realidad la estructura permite la posibilidad de otras llamadas. Por ejemplo, E puede llamar a D, a B y a A; C puede llamar a B y a A, etc. Por otra parte, la estructura de la Figura 7.1 impone algunas restricciones que pueden convertirse en indeseables. Por ejemplo, si descubrimos que el módulo F necesita la abstracción de procedimientos que proporciona el módulo E, la estructura de la Figura 7.1 es totalmente inadecuada. La Figura 7.2 nos presenta una reorganización de la estructura del programa que es compatible con esta nueva necesidad.

El problema en la organización de la Figura 7.2 es que no nos muestra la descomposición jerárquica de las abstracciones. El módulo E aparece como una abstracción subsidiaria utilizada por A, mientras que la única razón de esta situación en el árbol, es que los módulos B y F tienen que referirse a él.

Problemas análogos se dan con las variables (tipos y constantes). La estructura en árbol no prohíbe accesos indeseables a variables declaradas en módulos que las encierran. Además, si alguno de los módulos M_i y M_j deben compartir una variable, esta variable debe declararse en un módulo M que encierre estáticamente a M_i y M_j .

Más problemas surgen con la composición del texto de los programas Pascal. El programa completo es un sólo texto monolítico. Si el programa es grande, la separación de los módulos no es visible inmediatamente, aunque los programadores utilicen unas normas muy cuidadosas para su indentación. La parte de cabecera correspondiente a la declaración de constantes, tipos y variables del módulo, puede aparecer bastante antes de su cuerpo, por intervenir en esta cabecera, declaraciones de módulos internos. Como consecuencia, los programas, pueden ser difíciles de leer y de modificar.

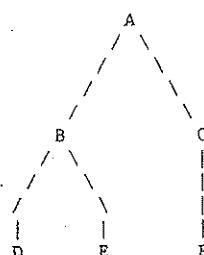


Figura 7.1 Árbol de anidamiento estático.

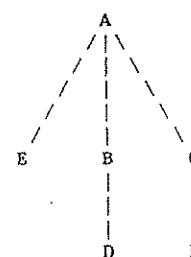


Figura 7.2 Un reajuste de la estructura del programa de la Figura 7.1

Los problemas de Pascal que se han tratado en esta sección, surgen de la estructura de bloque, y por lo tanto afectan a todos los lenguajes con una estructura similar a la de ALGOL. La estructura de bloque es adecuada para una programación a pequeña escala, ya que soporta el refinamiento paso a paso de forma natural. Esto no es válido para grandes programas, ya que la estructura del programa resultante del anidamiento, interfiere con las estructuras lógicas que se determinaron en la fase de diseño, lo cual deteriora la facilidad de escritura, la legibilidad y la modificabilidad de los programas.

7.3.1.3 Independencia Modular

Aquí veremos cómo se pueden desarrollar módulos independientemente, y qué vida y posibilidad de reutilización tienen estos módulos. Estos requisitos están parcialmente ligados con la etapa de diseño mediante una clara definición de las interconexiones entre módulos y una aplicación sistemática del principio de ocultamiento de información. Además, se desea soportar la implementación de módulos separados. Debería ser posible compilar y verificar cada módulo por separado, y a continuación poderlos guardar en una biblioteca para su posterior reutilización.

La descripción oficial de Pascal no define ninguna facilidad para poder realizar la compilación de módulos por separado. Sin embargo, varias implementaciones de este lenguaje, sí la soportan. Los módulos compilados por separado se pueden combinar para dar lugar a un único sistema en una fase lógicamente distinta. Seguir esta estrategia en la implementación de grandes programas, puede evitar muchos de los problemas ya tratados en las Secciones 7.3.1.1 y 7.3.1.2.

No obstante, la carencia en Pascal estándar de módulos de compilación separada, deja sin contestar algunos puntos importantes:

1. ¿Qué partes del programa puede exportar una unidad de compilación?
2. ¿Cómo se especifica la interconexión de una unidad?
3. ¿A qué nivel debe realizarse la comprobación de tipos entre las unidades interconectadas?

Distintas implementaciones adoptan diferentes soluciones para estos puntos. Como consecuencia de esto, programas Pascal desarrollados en distintas instalaciones pueden resultar incompatibles.

En la mayoría de las implementaciones actuales, sólo los procedimientos o funciones del nivel externo se pueden compilar separadamente. Las unidades de compilación separada se combinan más tarde, utilizando un montador proporcionado por el sistema operativo. El montador resuelve las ligaduras entre las entidades importadas por cada módulo y las correspondientes entidades exportadas por otros módulos. Desafortunadamente, el montador no comprueba la interconexión de módulos en cuanto a la corrección de tipos. Por ejemplo, una llamada a un procedimiento externo puede ser inconsistente con la declaración del procedimiento correspondiente y permanecer sin ser detectado. Se podría conseguir una facilidad de compilación separada más segura, como por ejemplo, en las líneas del esquema Ada que se trata más adelante, en la Sección 7.3.3.

7.3.2 SIMULA 67

7.3.2.1 Módulos

Los módulos lógicos en SIMULA 67 se pueden representar como procedimientos, funciones o clases. Aunque una clase puede agrupar un conjunto de declaraciones de datos y subprogramas relacionados, no proporciona ningún medio para ocultar detalles internos a otros módulos. El ocultamiento de información sólo se puede conseguir adoptando convenios disciplinados de programación.

7.3.2.2 Estructura del Programa

SIMULA 67 adopta la estructura de bloques convencional de tipo ALGOL. La característica poco usual del lenguaje es que los bloques y clases se pueden prefijar dando un nombre de clase. Prefijando una clase definimos una subclase (como se explicó en el Capítulo 4). Con el siguiente ejemplo se puede ver cómo se prefija un bloque. La siguiente clase, `proceso_de_listas`, define un conjunto de procedimientos que pueden usarse para manejar listas encadenadas.

```
class proceso_de_listas;
ref(elemento)primero;
boolean procedure vacia;
begin . . . comprueba si la lista esta vacia. . . end;
procedure insertar(e);
begin . . . inserta el elemento e en la lista. . . end;
procedure sacar(e);
begin . . . saca el elemento e de la lista. . . end;
begin
  primero := none
end
```

Al prefijar un bloque con el nombre de una clase, se consigue que los atributos de la clase sean visibles en el bloque. Por ejemplo, un bloque que empieza con

`proceso_de_listas begin . . .`

tiene acceso a los procedimientos insertar, sacar, vacia y a la referencia primero. Esta idea ha sido utilizada de una forma muy interesante para proporcionar un conjunto de facilidades normalmente usadas por los programadores de SIMULA 67. Las dos clases, SIMSET y SIMULATION, se han definido para realizar respectivamente el proceso de listas y las primitivas de simulación. Cualquier programa SIMULA 67 puede utilizar sus nombres como un prefijo. Estas dos clases pueden considerarse como bibliotecas fijas de módulos reutilizables.

El sistema de prefijos soporta el diseño modular descendente. Una clase de nivel superior puede escribirse de forma que contenga únicamente las decisiones más globales del diseño. Sucesivas subclases de esta clase contienen decisiones adicionales basadas en las anteriores, es decir, niveles inferiores de abstracción. En el nivel más bajo, el programa está prefijado por la clase más detallada.

La mayoría de los programas grandes (por ejemplo, sistemas operativos) existen en muchas versiones. Debemos poder controlar el desarrollo de todas las versiones, observando las similitudes y considerándolos como miembros de una familia de programas. En cada nivel de diseño, determinadas decisiones se hacen excluyendo algunos miembros de la familia. En el nivel más bajo, un miembro concreto permanece. Debido a que un cambio inicial en una decisión de diseño a un cierto nivel afecta a todas las decisiones de diseño de los niveles inferiores, un cambio en una decisión primaria de diseño, afecta al programa de una forma más sustancial que un cambio debido a una decisión posterior. Las subclases de SIMULA 67 soportan este método. Diferentes subclases de una clase representan distintos miembros de una familia con el mismo antecesor. Una clase engloba todas las decisiones de diseño comunes a todas sus subclases. Un cambio en una subclase no requiere ninguna modificación en la clase padre, pero si de las subclases de nivel inferior.

El problema básico con SIMULA 67 es que la prefijación está incluida dentro de la estructura de bloque. En consecuencia, las mismas críticas que se hicieron para Pascal en la Sección 7.3.1.2 sirven para el SIMULA 67.

7.3.2.3 Independencia Modular

Mediante la utilización disciplinada de las clases, se pueden diseñar programas utilizando la composición de módulos altamente independientes. Sin embargo, como en el caso de Pascal, la definición oficial de SIMULA 67 no especifica ningún tipo de convenio para el desarrollo con módulos separados.

La facilidad de compilación separada de módulos se ha introducido en la implementación de SIMULA 67 en el SYSTEM-10 de DEC. En esta implementación, las declaraciones de clases y procedimientos se pueden compilar separadamente. Cada módulo separado puede declarar una clase o un procedimiento como EXTERNAL, indicando la utilización de un módulo compilado por separado. Cada clase o procedimiento se debe compilar antes que cualquier módulo que haga referencia a él. Esto permite la comprobación de tipos en tiempo de compilación. Este esquema fuerza a un desarrollo ascendente del sistema, ya que los módulos que implementan abstracciones subsidiarias, se deben compilar antes que los módulos que utilizan esas abstracciones.

7.3.3 Ada

7.3.3.1 Módulos

Un módulo Ada puede ser un subprograma (genérico), un paquete (genérico) o una tarea. Los módulos lógicos identificados en la etapa de diseño, se pueden representar en módulos Ada de una forma bastante natural. Es más, el lenguaje soporta la distinción entre la especificación del módulo y el cuerpo de éste, permitiendo separar lo que exporta el módulo de lo que está escondido dentro de él.

7.3.3.2 Estructura del Programa

Un programa Ada es una colección lineal de módulos que pueden ser subprogramas (genéricos) o paquetes (genéricos). Estos módulos se denominan unidades. La unidad que implementa el programa principal es un subprograma en el sentido usual. Los módulos encierran declaraciones o módulos (más internos). En consecuencia, una unidad puede organizarse como una estructura de árbol. Cualquier abuso de anidamientos dentro de una unidad, genera los mismos problemas que se han discutido para Pascal.

Estos problemas pueden mitigarse utilizando la facilidad de la subunidad que ofrece el lenguaje. Esto permite que el cuerpo de un módulo incluya en la parte de declaración una unidad (o

subunidad) que se va a escribir separada de la unidad (o subunidad) que la encierra. En lugar de tener un módulo entero en la parte de declaraciones de la unidad que lo encierra, sólo es necesario que aparezca un cuerpo ficticio. El siguiente ejemplo nos muestra el concepto de subunidad.

```

procedure X(. . .) is -- especificación de la unidad
    W:INTEGER;
    package Y is -- especificación de la unidad interna
        A:INTEGER;
        function B(C:INTEGER) return INTEGER;
    end Y;
    package body Y is separate; -- cuerpo ficticio
begin -- utiliza el paquete Y y la variable W
    .
    .
end X;

separate (X)
package body Y is
    procedure Z(...) is separate; -- cuerpo ficticio
    function B(C:INTEGER) return INTEGER is
    begin
        -- utiliza el procedimiento Z
    .
    .
end B;
end Y;

separate (X.Y)
procedure Z(...) is
begin
    .
    .
end Z;
.
.
```

El prefijo **separate(X)** especifica que el cuerpo del paquete Y es una subunidad de la unidad X. Análogamente **separate (X.Y)** indica que el procedimiento Z es una subunidad del paquete Y anidado en X.

La subunidad no sólo puede aumentar la legibilidad de los programas, sino que también ofrece una técnica útil para la programación descendente. Cuando se está escribiendo un programa a un cierto nivel de abstracción, es mejor dejar que ciertos detalles se decidan en un nivel inferior. Supongamos que usted se da cuenta de que se requiere un determinado procedimiento para realizar una tarea determinada; aunque las llamadas al procedimiento pueden estar disponibles cuando quiera probar el flujo de ejecución, el cuerpo del procedimiento se puede escribir posteriormente.

Sin embargo, las facilidades proporcionadas por las

subunidades, no resuelven todos los problemas ocasionados por la estructura de árbol anidado. El cuerpo de una subunidad textualmente separada se considera que está lógicamente colocado en el punto en el que aparece su cuerpo ficticio en la (sub)unidad que lo encierra. Este punto es exactamente el que determina las entidades visibles a la subunidad. En el ejemplo anterior, ambas subunidades Y y Z pueden acceder a la variable W declarada en la unidad X.

Queda por ver el problema de las interconexiones (es decir, las relaciones de importación/exportación) entre las distintas unidades especificadas en Ada. Una unidad exporta todas aquellas entidades que se especifican en su parte de especificación. Puede importar entidades de otras unidades si y sólo si los nombres de dichas unidades se encuentran catalogados con una sentencia adecuada (sentencia with) que precede a la unidad.

Por ejemplo, la siguiente unidad cataloga X en su sentencia with. Por lo tanto está permitido llamar al procedimiento X dentro del cuerpo de T.

```
with X;
package T is
    C:INTEGER;
    procedure D(...);
end T;
package body T is
end T;
```

Análogamente, el siguiente procedimiento U puede llamar licitamente al procedimiento T.D y acceder a la variable T.C. Por el contrario, la unidad X no es visible por U.

```
with T;
procedure U(...) is
end U;
```

La interconexión de unidades en Ada se realiza a través de la sentencia with, la cual indica los nombres de las unidades de las que se pueden importar entidades, y a través de la especificación de la unidad, la cual lista las entidades que exporta la unidad. Cada módulo de la etapa de diseño se puede implementar como una unidad. Si se realizó cuidadosamente el diseño descendente, los módulos lógicos deberán ser relativamente simples. Consecuentemente, el nivel de anidamiento dentro de las unidades deberá ser muy pequeño, o incluso nulo. Sin embargo, nótese que Ada no prohíbe el abuso de anidamientos dentro de sus unidades. En realidad, un programa completo podría diseñarse como una única unidad con una estructura en árbol anidada en su

interior. Estructuras más deseables sólo pueden conseguirse mediante convenios para el desarrollo de programas, ya que el lenguaje no proporciona ningún medio para alcanzarlas.

7.3.3.3 Independencia Modular

La compilación del conjunto de unidades y subunidades que componen un programa, se puede realizar en una o más compilaciones separadas. Cada compilación traduce una o más unidades y/o subunidades. El orden de compilación debe satisfacer las siguientes restricciones.

- Una unidad se puede compilar si y sólo si todas las unidades mencionadas en sus sentencias with, han sido previamente compiladas.

- Una unidad se puede compilar si y sólo si la unidad que la encierra ha sido compilada previamente.

Además, las especificaciones de la unidad pueden compilarse separadamente del cuerpo. El cuerpo de una unidad se debe compilar después de que sus especificaciones hayan sido compiladas. La especificación de una unidad U mencionada en la sentencia with de la unidad W se debe compilar antes que W. No es así con el cuerpo de U, que puede compilarse antes o después de W.

Estas restricciones aseguran que una unidad se compile sólo después de que se tengan compiladas las especificaciones de las unidades de las que puede importar entidades. El compilador mantiene en un fichero de biblioteca los descriptores de todas las entidades exportadas por las unidades. Cuando una unidad es sometida a compilación, el compilador accede a la biblioteca, con lo cual pueden realizarse las mismas comprobaciones de tipo, tanto si el programa se compila en partes o como un todo. Si un paquete exporta un tipo de dato encapsulado (privado), la representación del tipo permanece oculta al programador, pero es conocida por el compilador, gracias a la cláusula private que aparece en la especificación del paquete. Consecuentemente, el compilador puede generar código para situar variables de tales tipos, declaradas en otras unidades que fueron compiladas antes que el cuerpo del paquete (pero después de su especificación).

Cuando una unidad se modifica, a veces es necesario recompilar otras unidades. El cambio puede afectar potencialmente tanto a sus subunidades como a todas las unidades cuyo nombre aparece en la instrucción with. En principio, se deben recompilar todas las unidades potencialmente afectadas.

La facilidad de compilación separada de Ada proporciona un desarrollo incremental de programas más que un desarrollo paralelo, ya que las unidades deben desarrollarse de acuerdo a un orden parcial. Esto no es una restricción arbitraria, sino una decisión de diseño consciente del desarrollo metódico de

programas. Una unidad se puede compilar sólo después de que las interconexiones con todas las unidades utilizadas estén firmemente especificadas. Por lo tanto, el programador está forzado a posponer el diseño del cuerpo de la unidad, hasta que estén diseñadas todas sus interconexiones.

Uno de los objetivos de la compilación separada es facilitar la producción de un software reutilizable. Los módulos que ya están verificados pueden guardarse en una biblioteca y posteriormente combinarlos para formar distintos programas. La solución de Ada a este respecto es deficiente para el caso de paquetes que exportan tipos de datos encapsulados (privados). La parte visible (especificación) de estos paquetes tiene que incluir las operaciones del tipo y una cláusula private que indica la representación interna del tipo. Esta representación no puede utilizarse fuera del cuerpo del paquete (su único propósito es facilitar la compilación separada). Lógicamente, esta información pertenece al cuerpo del paquete, junto con los cuerpos de los procedimientos que realizan las operaciones del tipo. Además de resultar estéticamente desagradable, esta característica tiene algunas consecuencias desastrosas:

- Viola el principio del diseño descendente. La representación tiene que determinarse al mismo tiempo que la especificación del tipo.
- Limita la potencia del lenguaje para utilizar las bibliotecas de módulos reutilizables. Por ejemplo, un módulo que utiliza colas FIFO, se compila y valida de acuerdo con un paquete de colas FIFO que proporciona una representación específica de colas FIFO (como puede ser un vector). El módulo tiene que recompilarse si se quiere utilizar en un programa distinto en el cual las colas FIFO utilizan una estructura de datos diferente, aunque las primitivas de manejo de las colas FIFO sean las mismas en ambos casos.

En este punto podemos contrastar la tentativa de Ada con el esquema existente en CLU. La facilidad que supone la compilación separada de CLU es similar a la de Ada. Sin embargo, CLU coloca los objetos de un tipo abstracto en la memoria libre. Los objetos se forman utilizando un procedimiento de creación encapsulado dentro del "cluster" que define el tipo abstracto. A los objetos creados se accede desde otros módulos utilizando punteros. Por lo tanto, los módulos no necesitan conocer la representación del tipo abstracto. Estos módulos sólo necesitan conocer las cabeceras de los procedimientos que crean y manipulan los objetos abstractos. Esta es la única información que es necesario que aparezca en la parte de especificación de un módulo, que implementa un tipo abstracto de datos. Un cambio de representación del tipo abstracto no requiere una recompilación de los módulos que utilizan tales objetos abstractos. En conclusión, CLU soporta mejor el desarrollo independiente que Ada.

7.3.4 Un Escenario Ideal

En las secciones previas hicimos hincapié en las diferencias conceptuales existentes entre la programación a gran escala y la programación a pequeña escala. Concretamente, en las Secciones 7.3.1 a la 7.3.3 se presentó una evaluación de las características de los lenguajes existentes que soportan la programación a gran escala. Este esquema es similar al que se tiene en Mesa.

Un lenguaje de programación debería componerse de dos sublenguajes: el lenguaje para la programación a pequeña escala (LPP) y el lenguaje para la programación a gran escala (LPG). Cuanto más claramente se distingan los dos niveles, más legibles serán los programas y más apropiado será el lenguaje para la programación de grandes sistemas. La mayoría de los lenguajes actuales no reconocen LPP y LPG como dos niveles separados, y por lo tanto los mezclan en la misma notación. Mesa sí lo hace, y proporciona un LPG denominado Lenguaje de Configuración Mesa (C/Mesa).

Como ejemplo para distinguir el LPP y el LPG, consideremos un módulo que importa algunos recursos externos y exporta determinados recursos internos (datos o procedimientos). La especificación de los recursos importados y exportados se indica en la interconexión del módulo escrito en LPP. Para los tipos abstractos de datos, solamente se permiten operaciones especificadas, no detalles de representación. La especificación de la interconexión permite que cada módulo LPP se pueda compilar independientemente de los demás módulos. Además, cada módulo puede hacer una comprobación completa de tipos con respecto a su interconexión. El LPP especifica los recursos importados, sin nombrar los módulos que los proporcionan. Los comandos de LPG establecen una correspondencia entre los recursos importados por cada módulo y los recursos exportados por otros módulos.

Por ejemplo, un módulo P que importa dos procedimientos P1 y P2 y exporta el procedimiento P3, puede tener la siguiente interconexión LPP.

```
import P1(integer,boolean);
       P2(integer,character);
export P3(real,real);
```

Un módulo Q con la siguiente especificación de interconexión

```
import . . .;
export Q1(integer,character);
```

Y un módulo R con la especificación de interconexión

```
import . . .;
export R1(integer,boolean);
```

se pueden estructurar en un sistema utilizando el siguiente

comando LPG:

```
bind P1 of P to R1 of R;
P2 of P to Q1 of Q;
```

El LPG es el responsable de la comprobación de tipos de la interconexión de cada módulo con respecto a los recursos proporcionados por otros módulos, y especificados en sus interconexiones. Es también responsable del montaje de un conjunto de módulos LPP validados en una única unidad ejecutable. El LPG puede verse como un lenguaje de control de trabajos que permite al programador especificar la interconexión entre los módulos que constituyen un sistema. Junto a las facilidades para especificar la configuración del sistema, un LPG debería proporcionar las facilidades de control de versiones y mantenimiento de las bibliotecas de descripciones de módulos. Ya que las descripciones evolucionan a medida que se desarrolla el sistema, el LPG debería soportar esta evolución. Esta sección trata básicamente al LPG como un lenguaje de configuración del sistema.

Los comandos LPG se ejecutan por un procesador que en parte es similar a un montador. Este procesador opera con módulos objeto. Para conocer si un recurso importado lo proporciona un módulo u otro, no es necesario volver a compilar ningún módulo, sino solamente realizar el proceso de LPG. Con esto se favorece la utilización de las bibliotecas de módulos objeto validados y reutilizables.

En este escenario ideal se plantea un problema en el caso de tener módulos que importan tipos abstractos de datos. Cuando estos módulos se compilan, el compilador ignora la representación de las variables locales de tipo abstracto.

Esta carencia de información no representa ningún problema en un lenguaje basado en la utilización de memoria libre, como es el caso del CLU, y esto es así porque a los objetos de un tipo abstracto los coloca en memoria el propio módulo que implementa el tipo abstracto, y otros módulos acceden a ellos sólo a través de punteros (ver Sección 7.3.3.4). En otros lenguajes, tales como Ada, los objetos de un tipo abstracto se deben colocar en memoria en la activación del módulo donde están declarados, pero las acciones en tiempo de ejecución necesarias para situar variables de un tipo abstracto importado, se conocen sólo después de que se procesa el LPG. Consecuentemente, el procesador LPG debe ser capaz de insertar tales acciones de tiempo de ejecución en los módulos objeto de LPP.

La anterior discusión nos muestra que un procesador LPG es considerablemente más complejo que un montador normal. Además de asociar nombres de recursos importados a los correspondientes recursos exportados, debe realizar la comprobación de tipos entre módulos, y para algunos tipos de LPP, generar código para la colocación en memoria de variables de tipo abstracto.

El hecho de que este escenario ideal requiera de una pieza extra (el procesador LPG), tradicionalmente sería razón suficiente para abandonar este esquema, pero tal y como estamos diciendo desde el principio, el objetivo final es conseguir un entorno formado por un lenguaje y otras herramientas para el desarrollo de software fiable. El procesador LPG no debería verse como una herramienta necesaria para soportar una característica del lenguaje, sino como algo capaz de soportar un método de diseño modular. Separar la función de montaje de la compilación, es lo mismo que separar la estructura del sistema de la implementación de los módulos.

La separación entre el compilador y el montador que resulta de este esquema es, en parte, una reminiscencia de la filosofía de lenguajes más antiguos, como FORTRAN y PL/I. Estos lenguajes, de hecho, tienen una facilidad de compilación separada, y las unidades compiladas por separado se ensamblan mediante un montador. Sin embargo, el montador es una herramienta de bajo nivel que en la mayoría de los casos no comprueba tipos en los enlaces entre los módulos. Usar un montador como un procesador LPG en el escenario ideal, significaría abandonar la seguridad de los tipos en el nivel de estructuración del sistema, cuando los errores resultan probablemente más costosos.

7.4 HERRAMIENTAS PARA EL DESARROLLO DE SISTEMAS

Este capítulo ha enfatizado las diferencias entre la programación a pequeña y a gran escala. En la Sección 7.3 se puso de relieve que la mayoría de los lenguajes no tienen facilidades específicas para soportar la programación a gran escala. En la Sección 7.3.4 se vio que un montador con chequeo de tipos es necesario para que el lenguaje nos ayude a distinguir entre los dos niveles de actividad. En teoría, por supuesto, no es necesaria tal facilidad, pues su utilidad reside en la ayuda que puede proporcionar para la realización de un software fiable. Esta sección examina los tipos de herramientas que se utilizan, junto con los lenguajes de programación, para la producción de software. Aunque estas herramientas también se utilizan en la programación a pequeña escala, su uso es esencial en un entorno de programación a gran escala.

La necesidad de tener herramientas se reconoció muy pronto en la era de la programación. De hecho, los ensambladores han sido una herramienta provechosa en la programación de los ordenadores (en lenguaje máquina). Actualmente se tiene una gran variedad de herramientas para el desarrollo, las cuales proporcionan un entorno adecuado para la programación, incluso en máquinas muy pequeñas tales como los ordenadores personales. Cuando se tiene que abordar la tarea de programar grandes sistemas, se ve que el lenguaje de programación por sí solo, no es suficiente. En este marco de trabajo, un programador individual ve al ordenador, no como un recurso personal, sino como un almacén de herramientas públicas y privadas. Algunas de las herramientas las proporciona el sistema; otras (los módulos

diseñados por otros programadores) se desarrollan exclusivamente para un proyecto específico. El ordenador es también el medio a través de cual fluye toda la información relevante. Se puede utilizar para guardar documentación, enviar mensajes y generar funciones de dirección y coordinación de proyectos.

Herramientas Básicas

A continuación se describen brevemente cuáles son las herramientas necesarias para el desarrollo de sistemas. Estas herramientas son básicas en el sentido de que son necesarias para la utilización de algún método de desarrollo. Posteriormente veremos algunas herramientas que soportan métodos específicos de diseño.

1. **Editor de textos** Es el componente del sistema que se utiliza para introducir programas (así como otros tipos de documentos) en el ordenador. El método más común de entrada de un programa es a través de un editor de textos interactivo, con lo cual se facilita la tarea de corregir el texto del programa.

2. **Macroprocesador** Es un simple traductor que en un documento fuente puede reemplazar una cadena específica por otra que se le indique. Macroprocesadores más sofisticados, permiten la parametrización de la cadena fuente, con lo que aumenta considerablemente su utilidad. Un macroprocesador se puede utilizar para:

(a) Superar alguna deficiencia del lenguaje, como puede ser la ausencia de constantes simbólicas. Por ejemplo, se puede utilizar TAMANO_MAXIMO en el programa y después, antes de compilar, decir al macroprocesador que lo sustituya por un valor específico.

De forma más general, se puede ampliar un lenguaje utilizando determinadas construcciones deseadas. Por ejemplo, RATFOR es un FORTRAN "estructurado" basado en este esquema, en el que unas cuantas estructuras (por ejemplo, if, for, while) se han definido como macros. Por lo tanto, un programa de RATFOR puede contener estas estructuras de control. Primero, el sistema RATFOR reemplaza estas macros por sus definiciones en FORTRAN, y después el programa se compila utilizando un compilador estándar de FORTRAN.

(b) Incrementar la legibilidad y facilidad de escritura de los programas. El uso de un macroprocesador puede ayudar a reducir los errores, dando un nombre a las partes de código que se repiten varias veces. Este nombre se puede usar a continuación en lugar de repetir el código.

Conforme se incrementa el tamaño de los segmentos de código repetido, se incrementa también la ventaja de esta técnica. Esta ventaja en la escritura puede traducirse en una mayor legibilidad, si se considera que la entrada al macroprocesador es el programa fuente. La secuencia de código que se repite varias veces en el programa resultante, es decir, la salida del macroprocesador, necesita leerse y entenderse sólo una vez en el programa fuente.

(c) Reemplazar llamadas a subprogramas con el cuerpo del subprograma. Así, puede diseñarse el programa como si cada unidad fuese un subprograma. Teniendo en cuenta la eficiencia del sistema, la decisión de cuándo implementar algo como un subprograma o insertar el código correspondiente en la llamada al subprograma, se puede posponer, y tomar la decisión más tarde, basándose en pruebas de ejecución. Ada permite este procedimiento proporcionando un directivo al compilador (**pragma INLINE nombre_subprograma**) que puede aparecer en la misma parte de declaración que el subprograma que nombra.

3. **Intérprete/Compilador** Un intérprete se usa para ejecutar un programa fuente. Normalmente puede producir un diagnóstico de errores mejor que un compilador, siendo por lo tanto más apropiado para la depuración del programa. Por otra parte, un intérprete es menos eficiente en la utilización del tiempo. Sería deseable tener un compilador y un intérprete que aceptasen el mismo lenguaje fuente, para que una vez que el programa esté validado, se utilice el intérprete para depuración y el compilador para dejar el sistema en explotación. El sistema INTERLISP incluye esta combinación para LISP.

4. **Sistema de ficheros** Un sistema de ficheros se utiliza para guardar datos y/o programas en formato fuente u objeto. El programador puede tener bibliotecas de componentes de programa, en el almacenamiento masivo. El sistema puede proporcionar ayuda para administrar esos componentes actualizando números de versiones, fechas de creación, etc. También es posible restringir el acceso a estos componentes a un conjunto específico de usuarios. Los programadores que trabajan en equipo pueden compartir sus componentes, y el sistema puede restringir a usuarios autorizados el acceso a tales componentes.

5. Editor de enlaces o Montador Un montador toma como entrada varios módulos previamente compilados y los mezcla en uno único de salida. La disponibilidad de un montador, tiene muchas ventajas: permite que distintos programadores trabajen concurrentemente en distintos módulos; y además permite compilar un módulo una sola vez y utilizarlo en la construcción de distintos sistemas. Para tener ventajas en un futuro, se puede crear una biblioteca de módulos reutilizables, que se podrán combinar de forma diferente. Finalmente, si se encuentra un error en un módulo, solamente tendremos que volver a compilar ese módulo. Estas son las ventajas que podrían obtenerse en un principio, dependiendo de las reglas del lenguaje y de la sofisticación del montador.

Estas son las herramientas básicas que hacen eficiente la utilización de un lenguaje de programación. Sin embargo, incluso con estas herramientas básicas se ve que existe una fuerte interdependencia entre el lenguaje, el método de diseño y las herramientas. Un buen ejemplo de esta interdependencia es el concepto de compilación separada en un lenguaje de programación, el cual requiere la utilización de un montador y una biblioteca como herramientas, y puede soportar un método basado en el diseño modular de software reutilizable.

Otras Herramientas

Las herramientas mencionadas anteriormente sirven para aumentar la productividad de la fase de codificación en el desarrollo de software. Existen otras herramientas para otras fases que tienen el propósito expreso de soportar un método de diseño específico.

Existe un número determinado de herramientas que soportan la fase de requerimientos, ayudando a llevar control de dichos requerimientos a la vez que los mantiene consistentes. Otra utilización de estas herramientas es la producción de sistemas de documentación. PSL/PSA (Teichrow 1975) y SREM (Alford 1977) son dos destacados ejemplos de estos sistemas. PSL/PSA proporciona un "lenguaje" (PSL) en el que se pueden expresar los requisitos, y un analizador (PSA) que examina su plenitud y consistencia. Debido a la diversidad de requisitos de los sistemas en diferentes áreas de aplicación, los llamados sistemas para requerimientos solamente son adecuados en ciertas aplicaciones. Por ejemplo, PSL/PSA es muy adecuado para sistemas de gestión, mientras que SREM lo es para sistemas en tiempo real.

Para soportar la fase de certificación, se dispone de los verificadores de programas, los cuales prueban automáticamente, o con ayuda del usuario, la corrección del programa con respecto a las especificaciones. Menos ambicioso que un verificador de programa, es un analizador de flujo de datos, el cual examina un programa estáticamente y prueba aquellos puntos que son fuentes

potenciales de error, tales como asignaciones sucesivas a una variable sin utilizar esa variable, utilización de variables sin inicializar, condiciones booleanas que siempre son falsas, etc. Los generadores de datos para pruebas ayudan en la generación de datos que sirven para probar el programa; los datos obtenidos deben guardarse para pruebas posteriores del programa. Los ejecutores simbólicos pueden ejecutar el programa con datos simbólicos, generando una serie de fórmulas simbólicas que caracterizan los valores de las variables de salida, y generando también condiciones para las rutas transversales del programa. Estas fórmulas las podrá utilizar el programador para razonar sobre su programa (seguir sobre el texto la lógica del programa, intentando descubrir errores).

Las herramientas de certificación deben conocer la sintaxis y la semántica del lenguaje con el que van a trabajar, ya que tienen una fuerte dependencia de éstas. Aquellas características que dificultan el seguimiento de la lógica de un programa (ver Capítulo 6), también complican el trabajo de las herramientas de certificación.

Una necesidad compartida por todas las herramientas de proceso de programas, es la de operar en la representación interna del programa. Si las herramientas van a estar integradas y deben trabajar juntas correctamente, todas deben utilizar la misma representación. Aunque estas necesidades restringen algunas herramientas utilizando un tipo de representación que puede no ser ideal, el beneficio de la compatibilidad de herramientas compensa esta desventaja. Una representación uniforme fomentaría también el desarrollo de programas compatibles, lo cual se pone de manifiesto en el sistema UNIX (Sección 7.4.1), en el que los ficheros se ven de forma uniforme, como una secuencia de caracteres que permite llevar la salida de un programa a distintos destinos, de un forma sencilla.

Otra motivación para el desarrollo de estas herramientas es soportar ciertos métodos de construcción de programas. Uno de estos métodos consiste en desarrollar una primera versión correcta del programa sin tener en cuenta la eficiencia de éste. Una vez terminado el programa, la siguiente tarea es modificarlo para que tenga una eficiencia aceptable. Los elementos necesarios para este método son los siguientes:

Analizador de frecuencia dinámica Ejecuta un programa con datos típicos, y determina qué porciones de programa se ejecutan más veces, indicando por lo tanto qué partes son las que necesitan una optimización mayor. Se ha demostrado que la intuición de los programadores es muy poco fiable para detectar estas áreas de código.

Optimizador de programas fuente Detecta y transforma código ineficiente en otro más eficiente.

Control de los programas fuente Permite mantener las distintas versiones de un programa y sus interrelaciones.

Estas diferentes versiones son especialmente importantes durante la fase de mantenimiento, ya que la versión original del programa, la que fue más fácil de programar, es también la más fácil de modificar.

7.4.1 Ejemplo de un Sistema de Desarrollo de Programas

Existen varios ejemplos de sistemas que combinan el lenguaje de programación con un conjunto de herramientas que proporcionan un entorno adecuado para desarrollar programas. APL y LISP disponen de este entorno. Esta sección considera el entorno de programación de UNIX. La razón de esta elección es que UNIX tiene una alta disponibilidad por lo que el lector tendrá fácil acceso a este sistema para experimentar.

UNIX es un sistema operativo para la familia de ordenadores DEC PDP-11. Es uno de los pocos sistemas operativos que proporcionan al usuario un entorno adecuado para desarrollar programas (aparte de otras cosas). El sistema está casi completamente escrito en C, lenguaje que también se utiliza como lenguaje oficial del sistema. Aunque C no es uno de los mejores lenguajes, el sistema cuenta con muchas herramientas que combinadas con C, proporcionan un excelente entorno para el desarrollo de programas. En parte, este hecho es el que ha dado la popularidad y el crecimiento del sistema UNIX. No es una exageración decir que lo mejor del lenguaje C es que viene con el sistema UNIX.

A continuación revisaremos algunas de las herramientas que proporciona UNIX para mostrar la integridad de un lenguaje de programación con un conjunto de herramientas que lo soportan.

Compilador C. El compilador tiene útiles e interesantes opciones. Una de ellas es el optimizador de código, que se utiliza después de que el programa ha sido depurado. Otra opción que se proporciona para el código objeto, es añadir contadores para dar una pista del número de veces que una rutina es llamada. Esto, como ya hemos visto, puede guiar al programador para determinar qué partes de programa son candidatas a futuras optimizaciones. También puede ayudar al programador a apreciar la perfección de la prueba, marcando las vías de control que han sido atravesadas durante la ejecución del programa. Finalmente, se puede generar un código objeto que posteriormente puede combinarse con otros módulos de código objeto (que posiblemente se escribieron en otros lenguajes).

Depurador C. Esta herramienta permite la ejecución y depuración interactiva de un programa. Con este sistema, el usuario puede seguir la ejecución de un programa, parar la ejecución en distintos puntos, y examinar los valores de las variables del programa. Una vez que el programa ha terminado se puede producir una traza y una indicación de la causa de fallo.

Lint. El compilador C no es bueno en la detección de errores o en la producción de mensajes explicativos de diagnóstico. La tarea de comprobar el programa se encarga al lint, que comprueba un programa C buscando secuencias de codificación sospechosa. El lint puede utilizarse con distintos propósitos. Uno es hacer cumplir las reglas de tipo de C. El compilador no hace una gran comprobación de tipos, y realiza libremente muchas conversiones de tipo. Por el contrario, el lint es más estricto, y marca aquellas secuencias de código que requieren conversiones automáticas. Con esto, el lint puede utilizarse para que parezca que el compilador de C hace una comprobación de tipos más fuerte.

El lint también puede utilizarse para la comprobación de tipos en las referencias intermodulares, con un conjunto de rutinas en distintos ficheros. De esta forma, programas grandes en C pueden dividirse en módulos, y cada módulo puede desarrollarse separadamente en cualquier orden. Mientras se compila un módulo, el compilador supone que las referencias externas de cada módulo son correctas. Una vez que se han desarrollado algunos o todos los módulos, se puede utilizar el lint para comprobar las compatibilidades de tipos. La división de labores entre el compilador y lint simplifica la del compilador, ya que éste puede concentrarse en la generación de un código eficiente y rápido, sin preocuparse de la comprobación de errores y la producción de mensajes de diagnóstico.

Otro propósito del lint es ayudar en la producción de programas transportables. Con esta opción, el lint señala aquellas sentencias del programa que denotan características no transportables, tales como comparaciones que dependen de la representación de caracteres de una máquina determinada.

El último tipo de comprobaciones realizadas por el lint son las construcciones propensas a error, entre las que se incluyen variables utilizadas antes de que se les asigne un valor, segmentos del programa no utilizados y expresiones booleanas cuyos valores son constantes en tiempo de compilación (siempre ciertas o siempre falsas). Estas condiciones existen originalmente en el programa o pueden aparecer después de un determinado número de modificaciones. En cualquier caso, al programador se le debe notificar su existencia. Es particularmente fácil durante el mantenimiento crear estas condiciones. Su existencia reducirá la legibilidad del programa, y como resultado, la facilidad de mantenimiento y su seguridad. Por lo tanto, el lint se debe aplicar después de cualquier modificación del sistema.

Make. Esta herramienta no está diseñada para el uso exclusivo de los programadores de C, sino que puede utilizarse en el desarrollo de cualquier sistema no trivial, en el que intervienen otros lenguajes. La razón de esta herramienta es que en la construcción de un gran sistema se

utilizan varios módulos, estos módulos se modifican individualmente y por lo tanto existen varias versiones. Aunque la tarea de ensamblar el sistema en una unidad ejecutable después de que varios módulos se han modificado, parece fácil en un principio, conlleva gran cantidad de tiempo, y con una alta tendencia al error. Se tienen que recordar los módulos que se deben compilar de nuevo, montar de nuevo, ejecutar a través del lint, qué bibliotecas hay que incluir y en qué orden, etc. El propósito del make es facilitar esta labor. El usuario especifica en un fichero todos los módulos que componen el sistema, sus interdependencias, y lo que hay que hacer con cada módulo (es decir, si hay que compilarlo en C o en FORTRAN, etc). Despues de que el usuario ha realizado alguna modificación en el sistema, se puede llamar a make para reagrupar los módulos utilizando sus últimas versiones, y ejecutar una secuencia de acciones definida por el usuario, pero esto sólo se hará con el mínimo número de módulos necesarios.

Esta herramienta se utiliza especialmente cuando varios programadores se agrupan para el desarrollo de un sistema y actualizan sus módulos independientemente.

Montador: El montador estándar, ld, se utiliza para combinar varios módulos objeto en uno solo.

Cref: Los listados de referencias cruzadas pueden realizarse con cref. Como entradas al cref pueden utilizarse tanto los programas escritos en C como en ensamblador.

Yacc y Lex: Son, respectivamente, un generador de analizadores sintácticos y un generador de analizadores léxicos. Su uso se ha confinado tradicionalmente al desarrollo de compiladores. Sin embargo, su utilización ha ido incrementándose en otras aplicaciones. Un programa que lee una entrada y comprueba su validez, necesita al menos un analizador léxico, que puede generarse utilizando el lex. Si además, el programador necesita leer e interpretar entradas complicadas, será necesario un analizador sintáctico, que puede generarse con el yacc. Por último, si el programa toma distintas acciones según las entradas recibidas (como es común en el caso de programas interactivos) la total generalidad del yacc puede utilizarse para organizar el programa como un traductor dirigido a la sintaxis: las acciones semánticas corresponden a las acciones que el programa necesita tomar para cada entrada. En este caso, el yacc dicta completamente la estructura del programa y genera automáticamente los analizadores léxicos y sintácticos. El programador necesita definir el lenguaje de entrada (la estructura sintáctica y el léxico de entrada) y las acciones para cada entrada.

Estas herramientas de generación de compiladores pueden verse como un caso especial de generadores automáticos de software, es decir, un sistema que genera

programas a partir de las especificaciones. La utilización de estas herramientas ayuda en la creación, validación y mantenimiento de programas, ya que lo único que se necesita hacer es cambiar la especificación.

Más importante que la disponibilidad de las anteriores herramientas, es la compatibilidad que existe entre ellas y con el sistema operativo. Por ejemplo, todos los ficheros tienen un formato uniforme, y las distintas herramientas conservan este formato, con lo que distintos editores de texto pueden utilizarse con el mismo fichero. Esto contrasta con otros sistemas, en los cuales cada editor necesita un formato distinto, con lo que si un fichero se edita con un editor, es imposible utilizar otro editor sobre él. Bajo UNIX, puede utilizarse un editor de texto distinto dependiendo de los requisitos del momento, sin tener en consideración la historia previa del fichero editado.

Otra gran ventaja de UNIX es la facilidad con que estas herramientas, y en general un conjunto de programas, pueden combinarse. El sistema operativo ofrece la facilidad de dirigir la salida de un programa a la entrada de otro a través de un sistema pipe. Por ejemplo, si A y B son dos programas,

A | B

denota la ejecución de A, utilizar su salida como entrada a B y ejecutar B (A y B se ejecutan concurrentemente mediante una relación productor/consumidor, pero esta implementación no la estudiamos aquí). Estas facilidades se utilizan mucho en el caso común en el que un programa necesita actuar de acuerdo con la salida de otro. Por ejemplo, ipr es un programa impresor que copia su entrada a una impresora de líneas. Si escribimos un programa P, cuya salida queremos que se imprima, todo lo que necesitamos decir es

P | ipr

Además, cuando escribimos P no necesitamos preocuparnos de en qué dispositivo va a escribirse la salida (como en FORTRAN); tampoco necesitamos llamar a un complicado procedimiento de control de trabajos que cree un fichero intermedio para que P ponga su salida en él, y luego en un paso separado, escribir los resultados de este fichero (como en el OS/360). En efecto, todos los programas se escriben suponiendo que la entrada es desde un dispositivo estándar de entrada (el terminal) y que la salida se dirige a un dispositivo estándar de salida (el terminal). Sin embargo, resulta bastante fácil cambiar los dispositivos de entrada/salida. Por ejemplo,

P < fe

indica que la entrada de P provenga del fichero fe, y

P > fs

indica que la salida de P se dirija al fichero fs (el cual se creará automáticamente si no existía anteriormente).

Estas facilidades estimulan el desarrollo de pequeños programas reutilizables, ya que cada programa escrito se puede ver como un componente potencial de un futuro sistema.

Los programas complejos no se crean escribiéndolos a partir de garabatos, sino conectando componentes relativamente pequeños. Estos componentes han de centrarse en funciones simples, y por lo tanto fáciles de construir, entender, describir y mantener. Experimentalmente se ha visto que el tamaño medio de un programa para UNIX escrito en C, es bastante pequeño (alrededor de 240 líneas).

No obstante, UNIX no es la respuesta final en sistemas de desarrollo. La mayor deficiencia de UNIX es que sus herramientas están especialmente dirigidas a las fases de codificación y pruebas. Lo que ahora se necesita, son herramientas para las fases de requisitos, validación y mantenimiento. Estas herramientas existen en casos aislados pero no se han integrado en un entorno tal como UNIX.

La importancia del entorno en la productividad del programador, ya está reconocida hoy día, y podemos esperar ver avances en este sentido. Por ejemplo, los requisitos para el lenguaje común del DOD especifican, no sólo el lenguaje (Ada), sino también el entorno completo de programación. Sin embargo, todavía nos quedan años para ver un sistema completo, coherente e integrado que soporte las facilidades del escenario idealizado en el primer capítulo de este libro (Sección 1.2).

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

La distinción entre la programación a pequeña y a gran escala se tiene en (DeRemer y Kron 1976). Las dificultades y desafíos que aparecen en la producción de grandes sistemas de software, se tratan en varios libros, tales como (Metzger 1973), (Brooks 1975) y (Horowitz 1975). Los escritos de H.D. Mills, L.A. Belady y M.M. Lehman, C.L. McGowan y R.C. McHenry en (Wegner 1979) estudian varios aspectos de la programación a gran escala.

El método de los refinamientos sucesivos se describe en (Wirth 1971b), (Wirth 1976a), y (Dijkstra 1972). Otro método mencionado en el Capítulo 6 intenta el desarrollo de un programa junto con sus pruebas de corrección. Dicho método se basa en la utilización de transformadores de predicados e invariantes, y se describe en (Dijkstra 1976). Ambos métodos son adecuados para el diseño de pequeños programas, pero tienen poca utilidad en el caso de programas grandes. Un interesante debate sobre este punto se tiene en (Wegner 1979). (Ver los escritos de D. Gries y el tratado de B. Liskov y D. Parnas).

Los métodos que soportan el diseño modular descendente se

describen en (Parnas 1972a), (Parnas 1972b), (Myers 1975) y (Myers 1978). El concepto de familias de programas se ve en (Parnas 1975).

Una evaluación crítica de la estructura en árbol de los grandes sistemas ha sido suscitada por (Wulf y Shaw 1973). (Clarke y otros 1980) hacen críticas más duras, y abogan por una estructura sin anidamientos en Ada.

(Lauer y Satterthwaite 1979) describen el sistema Mesa, y concretamente, cómo soportar el diseño de grandes sistemas. Smalltalk (Ingalls 1978) es otro lenguaje con interesantes facilidades de estructuración que está basado en SIMULA 67. PROTEL (Foxall y otros 1979) permite que un módulo tenga asociadas más de una interconexión. Esta técnica permite que usuarios distintos de un mismo módulo tengan distintos accesos a él. La experiencia con el uso del PROTEL, que es un lenguaje en el cual no tienen mucha importancia los tipos, se recogen en (Lasker 1979) y (Cashin y otros 1981).

Las facilidades para la compilación separada en Pascal se describen en (Jensen y Wirth 1975), (Kieburz y otros 1978), (Leblanc y Fisher 1979) y (Celentano y otros 1980). Las facilidades de compilación separada para el SIMULA 67 se describen en (Birtwistle y otros 1976) y (Schwartz 1978a). En (Birtwistle y otros 1973) se muestra con varios ejemplos la utilización del mecanismo de prefijo de SIMULA 67 en diseños descendentes.

La necesidad de sistemas para el desarrollo de software sofisticado se defiende en (Winograd 1979). (Cheatham 1977) analiza las herramientas para el desarrollo de programas y ve la necesidad de su integración en un sistema coherente. (Cheatham y otros 1979) presentan un sistema para el desarrollo de programas realizado en la Universidad de Harvard. (Tichy 1979) y (Habermann 1979) presentan otro sistema de la Universidad de Carnegie Mellon. (DOD 1980a) fija los requisitos para el entorno de programación de Ada. (Buxton 1980) ofrece una amplia bibliografía de trabajos en el área de sistemas para el desarrollo de programas. Para referencias sobre LISP y APL, ver el Capítulo 8.

El sistema UNIX se describe en (Ritchie y Thompson 1974). (Kernighan y Mashey 1979) describen el UNIX desde el punto de vista del usuario, derivado de la experiencia de un gran número de usuarios. El lenguaje C se describe en (Kernighan y Ritchie 1978). (Kernighan y Plauger 1976) describen las herramientas disponibles bajo UNIX.

Ejercicios

- 7.1 Discutir los efectos de las variables globales en la legibilidad y facilidad de escritura en los grandes programas.
- 7.2 Indicar las principales características de la compilación separada de FORTRAN.
- 7.3 ?Por qué la estructura de los programas de tipo ALGOL no es adecuada para la programación a gran escala?
- 7.4 Diseñar la interconexión de un módulo Ada que suministra una tabla de símbolos a un traductor (por ejemplo un ensamblador), y mostrar cómo procedimientos compilados separadamente pueden acceder a la tabla de símbolos. La estructura de datos que representa la tabla de símbolos debe esconderte del procedimiento, y todos los accesos a la tabla de símbolos deben realizarse a través de operaciones abstractas suministradas por el módulo de la tabla de símbolos. ?Puede compilarse el procedimiento antes de implementar una representación para la tabla de símbolos? ?Por qué? Si no se puede, ¿qué tiene de malo?
- 7.5 Supónganse dos unidades de Ada U1 y U2 que deben utilizar el mismo procedimiento P. ?Puede P estar incluido en una única subunidad? ?Puede P incluirse en una única unidad? En el último caso, ?cuáles son las restricciones en el orden de compilación?
- 7.6 Describir las herramientas de las que se debe disponer en un sistema ideal de desarrollo para soportar el desarrollo modular independiente, la estructura del sistema para lo anterior y una comprobación completa de tipos entre módulos.
- 7.7 Los dos conjuntos siguientes de comandos UNIX cumplen la misma función. ?Cuáles son las diferencias entre ellos?
 - i. x > fichero1
y < fichero1 > fichero2
z < fichero2
 - ii. x | y | z

Las letras x, y y z representan programas: fichero1 y fichero2 son ficheros.

PROGRAMACION FUNCIONAL



"... la próxima revolución en programación tendrá lugar solamente cuando se hayan cumplido los siguientes requisitos: (a) se haya desarrollado una nueva clase de lenguajes de programación, mucho más potentes que los de hoy día, y (b) se haya encontrado una técnica para ejecutar sus programas con un coste no mucho mayor que el de los programas de hoy día." (Backus 1978b)

La mayoría de los lenguajes que existen hoy día, incluyendo casi todos los que hemos discutido hasta aquí, pueden considerarse formados a lo largo de la misma línea de ideas. La arquitectura de Von Neumann es lo que caracteriza esta línea de desarrollo y dicta sus características esenciales. Todos los lenguajes que hemos estudiado hasta ahora, son abstracciones construidas sobre esta arquitectura.

Para proporcionar estas abstracciones, un lenguaje debe encontrar un equilibrio entre la utilización de características y la eficiencia de ejecución. La eficiencia de ejecución se mide por la realización en una máquina de Von Neumann. Así pues, la arquitectura de la máquina de Von Neumann ha constituido la base para el diseño de los lenguajes de programación.

El propósito de este capítulo es examinar otras bases para el diseño de lenguajes: las funciones matemáticas. El contraste entre estos dos fundamentos para el diseño de lenguajes nos ayuda a apreciar la influencia que tiene el modelo subyacente en el lenguaje.

8.1 CARACTERÍSTICAS DE LOS LENGUAJES IMPERATIVOS

En los primeros siete capítulos, nos hemos ocupado de los lenguajes imperativos, orientados a las sentencias. Esta clase de lenguajes, que contiene elementos tan diferentes como FORTRAN, COBOL, Pascal, CLU y Ada, debe su nombre al papel dominante que desempeñan las sentencias imperativas. La unidad de trabajo en un programa escrito en estos lenguajes es la "sentencia". Los efectos de las sentencias individuales se combinan en un programa para obtener los resultados deseados. Esta sección examina la estrecha relación entre los lenguajes y la arquitectura de los ordenadores convencionales. Veremos que la arquitectura del ordenador ha jugado un profundo papel en la formación de lenguajes imperativos.

Podemos ver esta influencia en tres características omnipresentes en nuestros lenguajes:

1. Variables. El principal componente de la arquitectura es la memoria, que consta de un gran número de celdas. La memoria es el lugar donde se almacenan los datos. Las celdas deben tener nombres, es decir, uno debe mantenerse al tanto de la colocación de la información.

La necesidad de dar nombres a las celdas, es más evidente cuando se programa en lenguajes de tipo ensamblador. Los valores se almacenan en celdas y se puede acceder a ellos dando nombre a dichas celdas. En los lenguajes de más alto nivel, la noción de celda de memoria y su nombre se representa por el concepto de variable, quizás el concepto más importante en los lenguajes de programación. Una variable en un lenguaje de programación es esencialmente una celda de memoria con nombre en cuyo interior se almacenan valores. Así pues, aunque nuestro propósito en programación sea producir valores, no podemos hablar solamente de los valores, sino también de las celdas en las que residen los valores de interés. Los problemas de efectos laterales y alias provienen de la existencia de variables.

2. Operación de asignación. Estrechamente ligada a la arquitectura de la memoria se encuentra la idea de que cada valor calculado debe ser "almacenado", es decir, asignado a una celda. Esta es la razón de la importancia de la sentencia de asignación en los lenguajes de programación. Las nociones de celda de memoria y asignación en bajo nivel, se extienden a todos los lenguajes de programación, y fuerzan en los programadores un estilo de pensamiento que se ha formado por los detalles de la arquitectura de Von Neumann.

3. Repetición. Un programa en un lenguaje imperativo normalmente realiza su tarea ejecutando repetidamente una secuencia de pasos elementales. Esto es una consecuencia de la arquitectura de Von Neumann, en la cual las instrucciones se almacenan en memoria. La única manera de llevar a cabo algo complicado es repitiendo una secuencia de instrucciones.

8.1.1 Un Programa Imperativo

Para ilustrar la interacción de estas características y sus efectos, consideraremos el siguiente programa Pascal para producir números primos:

```
(imprime números primos en el rango 2..n)
program primos (input, output);
  const n = 50;
  var i: 2 .. n;
      j: 2 .. 25;
      i_es_primo: boolean;
begin
  for i:= 2 to n do
    begin {?i es primo?}
      j:= 2; i_es_primo:= true;
      while i_es_primo and (j <= i div 2) do
        if ((i mod j) <> 0) then j:= j+1
          else i_es_primo:= false;
      (si lo es, imprime su valor)
      if i_es_primo then write (i)
    end
end.
```

El programa está basado en dos bucles, uno incluido dentro del otro. El bucle externo (for $i := 2$ to n ...) procesa los valores del rango de interés (desde 2 hasta n), mientras que el bucle interior examina si cada uno de los números es primo. Para entender cada bucle, debemos ejecutarlos mentalmente al menos para unas pocas iteraciones, debemos examinar sus condiciones de terminación, y debemos examinar las condiciones bajo las cuales se ejecutan correctamente. En este caso, el bucle interior depende (de una forma no muy simple) del "índice" del bucle externo (i). Este último también depende (mediante la sentencia if) de la asignación que se haga en el bucle interno.

En otras palabras, el programa no es jerárquico en el sentido de que cada componente esté compuesto por otros de nivel inferior. En cambio, cada componente usa los efectos de los otros. En nuestro caso, el bucle interno crea modificaciones en el bucle externo a través de i , y el bucle externo crea modificaciones en el bucle interno a través de i_{es_primo} . Estos componentes están internamente relacionados. Un componente se utiliza no sólo para producir un valor, sino también para producir un efecto, específicamente, el efecto de asignación de valores a las variables. Las estructuras de control se usan con el fin de que tales sentencias produzcan los efectos finales deseados.

8.1.2 Problemas de los Lenguajes Imperativos.

Resumiendo la sección anterior, la esencia de la programación con lenguajes imperativos es el cálculo paso a paso y de forma iterativa de valores de nivel inferior, y la asignación de estos valores a posiciones de memoria. Por supuesto, éste no es el nivel de detalle con el que nosotros queremos tratar cuando programamos una aplicación complicada. Claro está que los lenguajes de programación han sido un intento de ocultar más y más ese nivel natural de la máquina.

Un buen ejemplo de este intento lo constituyen las expresiones, que son un paso para alejarse de las celdas básicas de memoria, ya que permiten al programador mantener los valores intermedios anónimos. Las expresiones ocultan el hecho de que todas las celdas deben tener un nombre. En ausencia de efectos laterales, cada componente de las expresiones complejas es independiente de los demás componentes. FORTRAN proporciona esta ventaja sobre el lenguaje ensamblador, lo cual fue evidentemente uno de los mayores éxitos de FORTRAN. ALGOL 60 extendió las expresiones de FORTRAN proporcionando además, las expresiones condicionales. Por ejemplo, en vez de

```
if x>y then max := x
else max := y
```

podemos usar una expresión condicional y escribir

```
max := if x>y then x else y.
```

El término orientado a expresiones (como opuesto a orientado a sentencias) se refiere a lenguajes en los cuales las expresiones juegan un papel más importante que las sentencias. Ambos son términos relativos, y ningún lenguaje cuenta únicamente con expresiones o únicamente con sentencias. Podemos decir sólo que ALGOL 60 está más orientado a expresiones que FORTRAN.

ALGOL 68 está todavía más orientado a expresiones. Requiere que cada construcción sea una expresión, es decir, que tenga un valor que pueda usarse en una expresión. Un bloque, por ejemplo, puede ser usado en expresiones. Ejemplo:

```
max := begin int x,y;
read((x,y));
if x>y then x else y fi
end
```

El valor de la parte derecha de la asignación es x si $x > y$; y en caso contrario, su valor es y.

La utilidad de esta generalización de ALGOL 68 es questionable. Las expresiones se han encontrado útiles principalmente porque son simples y jerárquicas. Pueden combinarse uniformemente para construir expresiones complejas. Cada componente de una expresión compleja es independiente del resto de los componentes. Así pues, las expresiones no sufren influencias de la arquitectura de Von Neumann. Sin embargo, un bloque completo como expresión es bastante diferente, pues es susceptible de hacer las expresiones tan complicadas como cualquier otra construcción del lenguaje, porque una expresión puede estar compuesta de cualquiera de estas construcciones. En consecuencia, se necesitan reglas adicionales para explicar la semántica de las expresiones. En ALGOL 68 debemos saber que el último valor calculado en el bloque es el valor final que se le asigna al bloque. Si el valor del bloque no se usa (por ejemplo, siendo asignado) se aplica automáticamente un vaciado. Con una

construcción sobre una buena idea (las expresiones) se puede llegar bastante lejos. La generalización de ALGOL 68 hace también a las expresiones sujetos de problemas de denominación, repetición y asignación.

Realmente, hay dos dominios disjuntos en los lenguajes imperativos, el de las expresiones, que es simple, regular y jerárquico, y el de las sentencias. Es el mundo de las sentencias el que más influenciado está por las características de la máquina. El mundo de las expresiones también pierde sus efectos laterales o las construcciones que hemos tratado anteriormente.

Otro ejemplo de la interferencia de los detalles del nivel máquina con construcciones de lenguajes potentes se ofrece en el mecanismo de procedimiento. En los procedimientos se centran muchos de los razonamientos que deben hacerse sobre el comportamiento de las variables. Permiten la construcción de referencias y las variables globales interfieren con los conceptos de alto nivel permitiendo que todas las posiciones de memoria sean direccionables, y por lo tanto, revelando detalles de la memoria.

Quizás el problema más serio con los lenguajes imperativos surja de la dificultad de razonar sobre el grado de corrección del programa. Esta dificultad se produce por el hecho de que el grado de corrección de un programa en general depende del contenido de todas las celdas de memoria. El estado del cálculo está determinado por el contenido de las celdas. Para entender los bucles, debemos ejecutarlos mentalmente. Para observar el progreso de un cálculo a través del tiempo, debemos tomar "instantáneas" de la memoria en cada paso después de todas las instrucciones. Esta es una tarea pesada cuando el programa trata con grandes cantidades de memoria. Hemos visto que las reglas del ámbito en los lenguajes limitan algo este problema, reduciendo el número de celdas accesibles. También hemos visto (en los Capítulos 6 y 7) que el uso de variables globales puede hacer que un programa sea difícil de analizar.

Pero debemos recordar que mecanismos tales como variables globales, parámetros por referencia, y en general, los efectos laterales, que aparecen en contra de los propósitos de las construcciones de alto nivel, se introducen en los lenguajes con el propósito de alcanzar una eficiencia en la ejecución. Esta eficiencia está basada en las características de la arquitectura de Von Neumann. Y esto explica las razones de la creencia de Backus citada en el principio del capítulo.

La sección siguiente examinará el estilo de programación basado en las funciones y en las matemáticas, en vez de en la memoria de Von Neumann, las acciones elementales y la repetición. El propósito es ver qué alternativas existen si apartamos nuestra confianza de la arquitectura de Von Neumann.

B.2 LA ESENCIA DE LA PROGRAMACIÓN FUNCIONAL.

Mientras que el estilo de programación en lenguajes orientados a sentencias consiste en organizar la ejecución y repetición de sentencias individuales utilizando estructuras de control apropiadas, la esencia de la programación funcional es combinar funciones para producir funciones más potentes.

B.2.1 Funciones.

Una función es una regla de correspondencia (o asociación) de miembros de un conjunto (el dominio) con miembros de otro conjunto (el rango). Por ejemplo, la función "cuadrado" hace corresponder miembros del conjunto de los enteros con miembros del conjunto de los enteros. Una definición de función especifica el dominio, el rango y la regla de correspondencia para la función. Una vez se ha definido la función, puede ser aplicada a un elemento particular del dominio; esta aplicación produce (origina o devuelve) el elemento asociado dentro del rango.

Por ejemplo, la definición de función

$\text{cuadrado}(x) = x*x$, siendo x un número entero

define la función llamada "cuadrado" como una aplicación de los números enteros sobre los números enteros. Usamos el símbolo " $=$ " como "es equivalente a". En esta definición, x es un parámetro, y representa a cualquier miembro del dominio.

En el momento de la aplicación, se especifica un miembro particular del dominio. Este miembro, llamado argumento, reemplaza al parámetro en la definición, de forma puramente textual. Si la definición contiene cualquier otra aplicación, éstas son aplicadas de la misma forma hasta que tengamos una expresión que pueda ser evaluada para producir el resultado de la aplicación original.

La aplicación

$\text{cuadrado}(2)$

produce el valor 4 de acuerdo con la definición anterior.

El parámetro x es una variable matemática, lo cual no es lo mismo que una variable de programación. En la definición de la función, x representa cualquier elemento del dominio. En la aplicación se le da un valor concreto. Esto contrasta con la variable de programación que toma diferentes valores durante el curso de la ejecución del programa.

En la anterior definición de función, hemos asociado la definición de función $(x*x)$ con el nombre "cuadrado". Algunas veces conviene simplificar usando funciones sin darles un nombre. Podemos hacer esto usando las expresiones lambda. Una

expresión lambda especifica los parámetros y la regla de correspondencia. El símbolo ":" separa las especificaciones de parámetros de la regla de correspondencia. Por ejemplo,

$(\lambda x.x*x)$

es exactamente la misma función cuadrado anterior, excepto que no le hemos asignado un nombre. La función puede ser aplicada.

$(\lambda x.x*x)^2$

igual que "cuadrado".

El resultado de la aplicación se puede calcular reemplazando el parámetro y evaluando la expresión resultante.

$(\lambda x.x*x)^2 =$
 $2^2 =$
 4

Podríamos definir cuadrado como

$\text{cuadrado} = \lambda x.x*x$

En otras palabras, una expresión lambda nos permite construir una expresión cuyo valor es una función. Esta es una propiedad útil como veremos pronto.

La expresión lambda puede utilizarse para definir funciones de más de un parámetro, como en

$\text{suma} = \lambda x,y.x+y$

Los parámetros a continuación de " λ " y antes del ":" se llaman variables ligadas. Cuando se aplica la expresión lambda, las apariciones de estas variables en la expresión a continuación de ":" se reemplazan por los argumentos.

Se pueden crear nuevas funciones por combinación de otras funciones. La forma más común de combinar funciones en matemáticas es la función de composición. Si una función F está definida como la composición de dos funciones G y H , la representaremos como

$F = G \circ H$,

La aplicación de F se define como la aplicación de H , y luego, sobre el resultado, la aplicación de G . Por ejemplo, si definimos

$\text{a_la_cuarta} = \text{cuadrado} \circ \text{cuadrado}$

entonces el valor de $\text{a_la_cuarta}(2)$ es 16.

La función de composición es un ejemplo de forma funcional, también llamada forma de combinación o función de orden superior.

Una forma funcional proporciona un método de combinación de funciones; esto es, una función que toma otras funciones como parámetros y produce otra función como resultado. La composición de funciones es una forma funcional que toma dos funciones como parámetros y produce la función equivalente a aplicar una función y luego la otra como se ha expresado anteriormente.

8.2.2 Funciones Matemáticas Frente a Funciones de Lenguajes de Programación

Vamos a considerar las diferencias entre las funciones matemáticas y las que proporcionan los lenguajes de programación. La diferencia más importante proviene de la noción de variable modificable. El parámetro de una función matemática simplemente representa algún valor que se fija en tiempo de aplicación; la aplicación de la función origina otro valor. Por otro lado, los parámetros dato en las funciones de los lenguajes de programación son nombres de celdas de memoria; la función puede usar un nombre para cambiar el contenido de la celda. La función puede interactuar con el programa que llama a la función cambiando celdas conocidas por ambos. A consecuencia de este efecto lateral, las funciones de los lenguajes de programación, a diferencia de las matemáticas, no siempre pueden ser combinadas jerárquicamente (como expresiones).

Otra diferencia significativa es la forma en que se definen las funciones. En los lenguajes de programación, una función se define en forma de procedimiento: la regla de correspondencia de un valor del dominio con un valor del rango se establece en términos de un número de pasos que tienen que ser "ejecutados" en un orden específico mediante estructuras de control. Por otro lado, las funciones matemáticas se definen funcionalmente, es decir, la regla de correspondencia se define en términos de aplicación de otras funciones. En este sentido, las funciones matemáticas son más jerárquicas.

Muchas funciones matemáticas se definen recursivamente: esto es, la definición de la función contiene una aplicación de ella misma. Por ejemplo, la definición estándar matemática de factorial es

```
n! = if n=0 then 1 else n * (n-1)!
```

La secuencia de números de Fibonacci se define como

```
F(n) = if n=0 or n=1 then 1 else F(n-1)+F(n-2)
```

Como otro ejemplo, podemos formular fácilmente una solución recursiva para el problema de números primos de la Sección 8.1.1. La siguiente función predicado determina si un número es primo:

```
primo(n) = if n=2 then true else p(n, n div 2)
```

donde la función p es:

```
p(n,i) = if (n mod i)=0 then false  
else if i=2 then true  
else p(n,i-1)
```

La recursión es una potente técnica de resolución de problemas. Es una estrategia natural y ampliamente usada, cuando programamos con funciones. Sin embargo, como hemos visto en el Capítulo 3, no todos los lenguajes soportan la activación recursiva de subprogramas. Cuando no se permite la recursión, la implementación de funciones puede volverse complicada y contranatural.

8.2.3 Lenguajes Funcionales (o Aplicativos)

Un lenguaje de programación funcional hace uso de las propiedades matemáticas de las funciones. De hecho, el nombre "funcional" (o "aplicativo") se deriva del predominante papel que juegan las funciones y la aplicación de funciones. Un lenguaje funcional (o aplicativo) tiene cuatro componentes:

- * Un conjunto de funciones primitivas.
- * Un conjunto de formas funcionales.
- * La operación aplicación.
- * Un conjunto de objetos o datos.

Las funciones primitivas están predefinidas por el lenguaje, y pueden ser aplicadas. Las formas funcionales son mecanismos por los cuales se pueden combinar y crear nuevas funciones. La operación de aplicación es el mecanismo que se incorpora para aplicar una función a sus argumentos y producir un valor. Los objetos son miembros válidos del dominio y del rango de las funciones. Es característico de los lenguajes funcionales proporcionar un conjunto muy limitado de objetos con una estructura regular y simple (p.ej., matrices en APL y listas en LISP). La esencia de la programación funcional se encuentra en la combinación de funciones usando formas funcionales.

Además de los cuatro componentes de la relación anterior, los lenguajes funcionales también proporcionan mecanismos para dar nombre a las nuevas funciones que se definen. Esta es una característica conveniente que afecta a la forma en que nosotros usamos el lenguaje. Evita la repetición de la definición de la función cada vez que la función sea aplicada. Por ejemplo, en la Sección 8.2.1 definimos una función y le dimos por nombre "a_la_cuarta". Si no le hubieramos dado nombre, tendríamos que usar la definición

cuadrado^ocuadrado

en cualquier sitio que necesitásemos aplicarla.

La siguiente sección examina los lenguajes puramente funcionales e ilustra su utilización.

8.3 UN LENGUAJE FUNCIONAL PURO Y SIMPLE.

John Backus, el creador de FORTRAN, ha estado trabajando en lenguajes de programación puramente funcionales desde principios de la década de 1970. Esta sección presenta una visión simplificada del trabajo de Backus. En particular, llamaremos PF a un lenguaje sencillo para programación funcional.

Los objetos del lenguaje son bastante simples: un objeto puede ser un átomo o una secuencia (de objetos); la secuencia vacía se conoce como nil. Los átomos son secuencias de caracteres tales como A, ABC, 26.

Hay un conjunto de funciones y un conjunto de formas funcionales. No hay variables; todos los datos deben usarse literalmente.

PF es realmente una familia de lenguajes. Cada miembro proporciona una selección de funciones y formas funcionales. Backus mantiene que el lenguaje debería proporcionar "funciones primitivas potentes y de utilidad general, en vez de débiles que podrían usarse para definir otras útiles". La forma más funcional, es decir, la más "expresiva", que tiene un lenguaje después de las formas funcionales son los mecanismos para escribir programas.

Siguiendo la notación de Backus, representaremos una secuencia de n objetos x_1, x_2, \dots, x_n como

$\langle x_1, x_2, \dots, x_n \rangle$

y una aplicación de la función f sobre el parámetro x como

$f:x$

Usaremos la expresión condicional if_then_else en las definiciones. Esto es una parte de nuestro lenguaje de definición, no parte del lenguaje que se está definiendo. Recuérdese que este lenguaje no está completamente definido y que es exactamente eso lo que queremos hacer.

8.3.1 Funciones Primitivas

El lenguaje proporciona un cierto número de funciones primitivas que a grosso modo se corresponden con las operaciones predefinidas en los lenguajes imperativos. Dado que las secuencias son objetos manipulables, las funciones proporcionan mecanismos para la manipulación de secuencias. Estas funciones pueden dividirse en varias clases diferentes. A continuación se da esta clasificación junto con algunos ejemplos de funciones primitivas. Desde luego, un lenguaje en concreto puede incluir un conjunto de funciones diferentes.

a. Operaciones de selección

Como ejemplos útiles se incluyen las funciones FIRST, para extraer el primer elemento de una secuencia; LAST, para extraer el último elemento; y TAIL, para extraer todos los elementos excepto el primero. Así pues,

FIRST: $\langle x_1, x_2, \dots, x_n \rangle = x_1$

LAST: $\langle x_1, x_2, \dots, x_n \rangle = x_n$

TAIL: $\langle x_1, x_2, \dots, x_n \rangle = \langle x_2, \dots, x_n \rangle$

También podría ser útil una función que fuera capaz de seleccionar de forma arbitraria un elemento de una secuencia simplemente dando su posición, es decir,

i: $\langle x_1, \dots, x_n \rangle = x_i, 1 \leq i \leq n$

b. Operaciones con estructuras

Estas operaciones nos permiten combinar, analizar o reorganizar secuencias.

(rotar a la derecha) ROTR: $\langle x_1, \dots, x_n \rangle = \langle x_n, x_1, \dots, x_{n-1} \rangle$

(rotar a la izquierda) ROTL: $\langle x_1, \dots, x_n \rangle = \langle x_2, x_3, \dots, x_n, x_1 \rangle$

(longitud) LENGTH: $\langle x_1, \dots, x_n \rangle = n$

(construir una secuencia) CONS: $\langle x, \langle x_1, \dots, x_n \rangle \rangle = \langle x, x_1, \dots, x_n \rangle$

c. Operaciones aritméticas

Las operaciones aritméticas se aplican sobre secuencias de dos átomos y producen un átomo como resultado. Consideremos las operaciones usuales "+", "-", "*", "/", "div" y también la operación resto ("mod" en Pascal) "|". Es decir,

|: $\langle x, y \rangle$

produce el resto de la división entera " x div y ". El signo unitario menos se designará por NEGATE. Conservando la notación funcional usamos la forma prefija para todas las operaciones aritméticas (p.ej., +: $\langle x, y \rangle$ en vez de la notación infija usual ($x+y$)).

d. Funciones predicados

Las funciones predicado son aquellas funciones cuyo resultado es un valor de verdad o falsedad. Representaremos cierto por el átomo T y falso por el átomo F. Así pues, una función predicado produce un valor F o T. Además de los predicados ya familiares de

comparación de números (p. ej., $<$, $>$, $=$) también necesitamos predicados para preguntar por secuencias y átomos.

```
ATOM:x = if x es un átomo then T else F
NULL:x = if x=nil then T else F
```

e. Operaciones lógicas

Las operaciones lógicas nos permiten combinar valores de verdad o falsedad. Las operaciones usuales son: AND, OR, NOT.

f. Identidad

La función ID es la función identidad, es decir,

```
ID:x = x
```

8.3.2 Formas Funcionales

Realmente la característica menos usual de los lenguajes PF es la forma funcional. Dado que no usamos formas funcionales en los lenguajes convencionales, es difícil decidir cuáles de las formas funcionales son de utilidad para incluirse en el lenguaje PF. También es difícil dar utilidad a las formas funcionales como construcciones de programación. Pero únicamente, como en los lenguajes convencionales, hasta que se adquiere práctica.

A continuación, examinamos unas cuantas formas funcionales que Backus consideró útiles como punto de partida, con el propósito de introducir al lector en la potencia de la programación con formas funcionales.

a. Composición

$$(f \circ g):x = f:(g:x)$$

La forma funcional " \circ " se define tomando como parámetros dos funciones; su resultado es una función equivalente a la aplicación del primer parámetro sobre el resultado de la aplicación del segundo parámetro. En otras palabras, es la función matemática de composición. Ya hemos visto un ejemplo de su utilización en la definición de "a_la_cuarta" en la Sección 8.2.1.

Para encontrar el valor de la aplicación de una función que es composición de otras funciones, repetimos cada paso de aplicación como antes, hasta que no quedan más aplicaciones en la expresión. En general, las aplicaciones más internas se ejecutan primero, luego las del siguiente nivel exterior, etc.

Ejemplo

```
ROTL°CONS:<x1, <x2, x3>> =
ROTL:CONS:<x1, <x2, x3>> =
ROTL:<x1, x2, x3> =
<x2, x3, x1>
```

Es realmente sorprendente que a pesar de la importancia y el predominio de la función composición en matemáticas, ningún lenguaje de programación la permite como una construcción primitiva, ni explícita ni tácitamente.

b. Construcción

$$[f₁, f₂, ... f_n]:x = <f₁:x, ... f_n:x>$$

La forma funcional "[]" se define tomando como parámetros n funciones y produce una función equivalente a aplicar cada una de las funciones sobre el mismo parámetro y formar una secuencia con los resultados.

Como ejemplo del uso de la construcción, consideraremos la tarea de conseguir el mínimo, el máximo, el promedio y la mediana de una secuencia de valores. Teniendo definidas funciones individuales para cada una de las subtareas, la tarea final de combinarla se puede hacer con la construcción, por ejemplo,

$$[\text{MINIMO}, \text{MAXIMO}, \text{MEDIA}, \text{MEDIANA}]$$

es una función que puede ser aplicada o combinada con otras funciones.

Ejemplo

```
[MINIMO, MAXIMO, MEDIA, MEDIANA]: <0, 1, 2, 3> =
<MINIMO: <0, 1, 2, 3>, MAXIMO: <0, 1, 2, 3>,
MEDIA: <0, 1, 2, 3>, MEDIANA: <0, 1, 2, 3>> =
<0, 3, 1.5, 2>
```

Nótese que en el segundo paso tenemos una secuencia formada por cuatro aplicaciones. Dado que no existen efectos laterales, estas aplicaciones pueden ser evaluadas en cualquier orden, o incluso simultáneamente.

c. Inserción

```

/f:x = if x es <x1> then x1
      else if x es la secuencia <x1...xn>
           and n>2
           then f:<x1, /f:<x2, ...xn>>

```

La forma funcional "/" toma como parámetro una función y produce como resultado otra función que sólo es aplicable a secuencias (no a átomos) y es equivalente a aplicar la función parámetro a los sucesivos elementos de la secuencia. El uso más evidente de "/" es en la distribución de funciones definidas para dos parámetros sobre una secuencia de cualquier número de elementos. Por ejemplo, suponiendo que la función primitiva de adición ("+") haya sido definida para aplicarse sobre dos parámetros, podemos encontrar la suma de una secuencia de cualquier longitud con la función "/+".

```

/+:<1,2,3,4> - +:<1, /+<2,3,4>
- +:<1,+<2, /+<3,4>>>
- +:<1,+<2,+<3, /+<4>>>>
- +:<1,+<2,+<3,4>>>
- +:<1,+<2,7>>>
- +:<1,9>
- 10

```

Este ejemplo muestra la potencia de la programación con formas funcionales. Consideremos la forma escueta y simple en que hemos definido la función suma "/+" y contrastémosla con la forma en que deberíamos haberla escrito en un lenguaje imperativo.

d. Constante

Esta forma funcional toma un objeto (x) como parámetro y produce una función.

```
x:y = x
```

Por ejemplo,

```

0:y = 0
T:y = T
1:y = 1

```

Más adelante, en el apartado g, se da un ejemplo del uso de esta forma funcional.

e. Aplica a todo

```

af:x = if x es nil then nil      if      nil then nil
      else if x es la secuencia <x1, x2, ... xn>
           then <f:x1, ... f:xn>

```

La forma funcional "a" toma como parámetro una función y produce una función, aplicable sólo a secuencias, que es equivalente a aplicar la función parámetro a cada uno de los elementos de la secuencia y formar una secuencia con los resultados.

Como ejemplo del uso de "a", consideraremos el siguiente problema. Tenemos una secuencia de secuencias. Cada una de las secuencias interiores contiene dos átomos. Queremos producir una secuencia, cada uno de cuyos elementos sea la suma del par de elementos en la secuencia del argumento. Por ejemplo, aplicada al argumento <<1, 2>, <3, 4>, <5, 6>>, nuestra función debería producir <3, 7, 11>.

La tarea puede realizarse simplemente con la función

a+

(Véase Ejercicio 1 para una generalización de ésta función y Ejercicio 3 para un ejemplo de su utilización).

Igual que la forma funcional inserción, aplicar a todo señala el contraste entre programación imperativa y funcional. Si una misma operación debe ejecutarse sobre cada uno de los elementos de una secuencia, un programador que usa un lenguaje imperativo debe diseñar la operación, luego diseñar un "bucle" para aplicar la operación a cada elemento. Diseñar el bucle entraña el manejo de asuntos como el acceso a cada elemento y la detección y manejo del final de la secuencia, de la secuencia vacía, etc. En un lenguaje de programación funcional, sin embargo, el diseño de la operación en sí mismo es todo lo que se necesita; aplicar a todo se encarga del bucle.

f. Condición

```
(IF p f g):x = if p:x=T then f:x else g:x
```

La forma funcional IF toma tres funciones como parámetros y produce la segunda función, o bien la tercera, dependiendo de si el resultado de la primera función es el átomo T o no lo es.

Esta forma funcional se puede utilizar en tareas en las que con lenguajes imperativos se usaría la sentencia condicional, pero debido a que p, f y g se aplican todas sobre el mismo parámetro, su uso requiere un enfoque diferente. Por ejemplo, la función

```
(IF ATOM NEGATE RÖTR)
```

niega su argumento si es átomo, o lo rota a la derecha

si es una secuencia.

g. Mientras

```
(WHILE p f):x = if p:x=T then (WHILE p f):(f:x)
else x
```

La forma funcional WHILE produce una función que realiza la aplicación repetida de su segundo parámetro tantas veces como la aplicación de su primer parámetro produzca el átomo T.

Por ejemplo, escribamos una función que produzca la imagen de la secuencia argumento, excepto los ceros no significativos. Primero definimos la siguiente función ESCERO, la cual determina si un átomo es cero o no lo es.

```
=:[ID,0]
```

Luego definimos la siguiente función PRIMEROESCERO, la cual determina si el primer elemento de una secuencia es cero o no lo es:

```
(ESCERO-FIRST)
```

Finalmente, definimos la función deseada como

```
(WHILE PRIMEROESCERO TAIL)
```

Los siguientes pasos iniciales de la aplicación de esta función nos ayudan a entender cuantas "ejecuciones" están implícitas en la ejecución de una función aparentemente tan sencilla.

```
(WHILE PRIMEROESCERO TAIL):<0,0,2>
if PRIMEROESCERO:<0,0,2>=T
then (WHILE PRIMEROESCERO TAIL):(TAIL:<0,0,2>)
else <0,0,2> =
(WHILE PRIMEROESCERO TAIL):<0,2>
if PRIMEROESCERO:<0,2>=T
then (WHILE PRIMEROESCERO TAIL):(TAIL:<0,2>)
else <0,2> =
(WHILE PRIMEROESCERO TAIL):<2> = . . . = <2>
```

8.4 CARACTERISTICAS FUNCIONALES EN LOS LENGUAJES EXISTENTES

La sección anterior ha proporcionado una visión rápida de los lenguajes PF. Estos lenguajes puramente funcionales han sido estudiados como base para una nueva generación de lenguajes de programación. No son lenguajes de programación cotidianos, sino que nos proporcionan un modelo para el estudio de los beneficios potenciales e inconvenientes de la programación funcional, a la

vez que también pueden ayudarnos a evaluar las características de los lenguajes existentes. Ya hemos discutido bastante la característica funcional o no-funcional de las expresiones. El propósito de esta sección es examinar otras características importantes de los lenguajes funcionales actuales.

8.4.1 Lisp

De todos los lenguajes de programación existentes, LISP (List Processing) viene a ser el más similar a un lenguaje funcional. En efecto, el LISP original introducido por John McCarthy en 1960, conocido como LISP simplemente, es completamente funcional. Sin embargo, con el fin de proporcionar una ejecución eficiente, las versiones actuales de LISP han introducido características no funcionales en el lenguaje.

Esta sección mostrará por qué el LISP es un lenguaje funcional. No intentaremos proporcionar una discusión completa, ni siquiera una introducción a LISP, sino simplemente a sus características funcionales. LISP es un buen ejemplo de cómo unas pocas primitivas junto con una facilidad de estructuración de datos simple pero potente puede producir un lenguaje elegante y potente.

8.4.1.1 Objetos

Los objetos en LISP son expresiones simbólicas que representan átomos o listas. Un átomo es una cadena de caracteres (letras, dígitos y otros). Los siguientes son ejemplos de átomos:

A
SYNAPSE
MC68000

Una lista es una serie de átomos o listas, separadas por blancos y encerradas entre paréntesis. Los siguientes son ejemplos de listas:

(ALIMENTOS VEGETALES BEBIDAS)
((CARNE POLLO) (COLIFLOR PATATAS TOMATES) (AGUA))
(UNC TRW SYNAPSE)

La lista vacía "()" también llamada NIL, tiene un significado especial. Como la secuencia en los lenguajes PF, la lista es el único mecanismo de estructuración de los datos para la información codificada en LISP.

Un programa LISP es en sí mismo una lista. Es funcional, ya que está compuesto de aplicaciones de funciones que producen resultados que se pueden usar en otras funciones. Incluso la notación es funcional, es decir, prefija, en oposición a la notación infija de otros lenguajes (p. ej.: (PLUS A B) en vez de

8.4.1.2 Funciones

En LISP puro se proporcionan muy pocas funciones primitivas. Existen sistemas LISP que han aumentado considerablemente el número de éstas. Sin embargo, estas nuevas funciones se pueden expresar en términos de las funciones primitivas originales:

QUOTE es la función de identidad. Devuelve como valor su propio argumento simple. Esta función es necesaria, ya que, al contrario de los PF, el átomo A no se representa a sí mismo, sino que es el nombre de un valor almacenado en algún lugar. La distinción entre el nombre y el valor de los objetos, como hemos visto, es una consecuencia directa de la arquitectura de la memoria. La función QUOTE permite que su argumento sea tratado como una constante. Así pues, (QUOTE A) en LISP es análogo a "A" en lenguajes convencionales.

Ejemplos

```
(QUOTE A) = A
(QUOTE (A B C)) = (A B C)
```

Las funciones más comunes son aquellas que manipulan las listas, aquellas que hemos llamado operaciones de estructuración.

CAR devuelve el primer elemento de una lista; CDR devuelve todos los elementos de una lista excepto el primero; CONS añade un elemento a una lista. Ejemplo:

```
(CAR (QUOTE (A B C))) = A
```

El argumento necesita llevar QUOTE porque la regla en LISP es que una función se aplica a los valores de sus argumentos. En nuestro caso, la evaluación del argumento produce la lista (A B C), sobre la cual se aplica CAR; si faltase QUOTE, se habría hecho un intento de evaluar (A B C), lo cual significaría la utilización de A como una función que opera sobre los argumentos B y C. Si A no se hubiera definido previamente, se produciría un error.

Otros ejemplos

```
(CDR (QUOTE (A B C))) = (B C)
(CDR (QUOTE (A))) = () - NIL
(CONS (QUOTE A) (QUOTE (B C))) = (A B C)
(CONS (QUOTE (A B C)) (QUOTE (A B C))) = ((A B C) A B C)
```

También se dispone de unos pocos predicados. El valor cierto se representa por el átomo T y el valor falso por NIL.

ATOM comprueba su argumento para ver si es un átomo.

NULL comprueba su argumento para ver si es NIL.

EQ comprueba la igualdad de sus dos argumentos, los cuales deben ser átomos.

Ejemplos

```
(ATOM (QUOTE A)) = T
(ATOM (QUOTE (A))) = NIL
(EQ (QUOTE A) (QUOTE A)) = T
(EQ (QUOTE A) (QUOTE B)) = NIL
```

La función COND toma como argumentos una serie de pares (predicado, expresión). La expresión del primer par (de izquierda a derecha) cuyo valor sea cierto será el valor del COND.

Ejemplo

```
(COND ((ATOM (QUOTE (A))) (QUOTE B))
      (T (QUOTE A))) = A
```

La primera condición es falsa porque (A) no es un átomo. La segunda condición es exactamente el valor cierto. Esta función, conocida como la condicional de McCarthy, es la mejor forma de construcción de bloques en las funciones definidas por el usuario.

La definición de funciones se basa en las expresiones lambda. La función:

$\lambda x, y. x+y$

se escribe en LISP como

```
(LAMBDA (X Y) (PLUS X Y))
```

también se permiten las expresiones lambda en la aplicación de funciones.

```
((LAMBDA (X Y) (PLUS X Y)) 2 3)
```

liga a X e Y con 2 y 3 respectivamente, y aplica PLUS produciendo el valor 5.

La asignación de un nombre a la función se hace mediante la función DEFINE.

```
(DEFINE (SUMA (LAMBDA (X Y) (PLUS X Y))))
```

Ahora se puede usar el átomo SUMA en lugar de la función anterior: es decir, el valor del átomo SUMA es una función. La utilización de DEFINE es uno de los dos caminos en LISP puro mediante los cuales un átomo puede ligarse con un valor. La asignación convencional no está presente aquí.

Las variables en LISP puro son más parecidas a las variables matemáticas que a las de los otros lenguajes. En concreto, las variables no pueden ser modificadas, se pueden ligar con un valor y retenerlo a través de un ámbito determinado (p.ej., la aplicación de una función); y en cualquier momento, hay a lo sumo

una única vía de acceso a cada variable.

8.4.1.3 Formas Funcionales

La composición de funciones era la única técnica de combinación de funciones que proporcionaba el LISP original. Por ejemplo, la función "a_la_cuarta" de la Sección 8.2.1 se puede definir en LISP como

```
(LAMBDA (X) (CUADRADO (CUADRADO X)))
```

Asumiendo que CUADRADO se ha definido previamente. Sin embargo, los sistemas LISP más recientes ofrecen una forma funcional, llamada MAPCAR, que es la equivalente a " α " en los PF. Permite la aplicación de una función a cada elemento de una lista. Por ejemplo,

```
(MAPCAR ALACUARTA L)
```

eleva a la cuarta potencia cada elemento de la lista L. Igual que α , esta forma funcional reduce la necesidad de repeticiones.

Usando las características funcionales de LISP ya citadas, podemos escribir el programa de números primos directamente a partir de la función vista en la Sección 8.2.2. Esto es lo que se pedirá en el Ejercicio número 10.

Como lenguaje funcional, LISP está bastante limitado por la falta de un amplio conjunto de formas funcionales.

8.4.1.4 Características no Funcionales de LISP

Hasta ahora, hemos considerado las características funcionales de LISP. Pero aunque es posible usar LISP como un lenguaje funcional, muy pocas aplicaciones se escriben en LISP de una forma puramente funcional. Consideraciones de eficiencia han forzado la introducción de muchas características no funcionales en LISP y cualquier programa realista hace un amplio uso de estas características para conseguir un nivel razonable de eficiencia.

Las principales características no funcionales añadidas a LISP son SET y PROG. La función SET es simplemente la sentencia de asignación. PROG es una función que toma una lista de expresiones como argumento, las cuales se ejecutan una tras otra. En otras palabras, PROG permite que las expresiones se manejen como sentencias. Además, con PROG se pueden usar etiquetas y sentencias goto para el control explícito de la secuencia.

Con la introducción de operaciones de modificación de variables, LISP se hace todavía menos funcional. Por ejemplo, la operación RPLACA (reemplazar CAR) se puede usar para reemplazar el primer elemento de una lista por otro, es decir, por razones

de eficiencia se introducen operaciones que trabajan mediante efectos laterales en vez de produciendo valores.

Incluso LISP puro muestra efectos provocados por la arquitectura de la máquina en la que se ha diseñado. El más importante de estos efectos aparece, como hemos visto, en la distinción entre nombres y valores. Otro de ellos aparece en el conjunto de funciones proporcionadas. La razón de que se proporcionen los dos selectores CAR y CDR, pero no se proporcionen, por ejemplo, selectores de un elemento arbitrario de la lista o del último elemento, es la forma en que está implementada la lista. Debido a la implementación, CAR y CDR se pueden ejecutar con referencia a memoria, mientras que los otros no.

Hemos examinado LISP solamente en un ámbito limitado. LISP es un lenguaje potente e interesante con un gran futuro. Es el lenguaje escogido para el desarrollo de sistemas experimentales y el lenguaje estándar en inteligencia artificial. Para una mayor información consultense las referencias citadas en la Ampliación Bibliográfica.

8.4.2 APL

APL fue diseñado por Kenneth Iverson en la Universidad de Harvard a finales de la década de los 50 y principios de los 60. Está basado en las matemáticas, con unas pocas concesiones a la eficiencia de la máquina. La operación de asignación es parte integrante del lenguaje. No obstante, se puede ver a APL como un lenguaje funcional debido a que su mayor importancia reside en las expresiones.

Igual que hicimos con LISP en la sección anterior, aquí examinaremos las características funcionales de APL, sin tratar de abarcarlo por completo.

8.4.2.1 Objetos

Los objetos que soporta APL son escalares, que pueden ser números o caracteres, y matrices de cualquier dimensión. Los números 0 y 1 se pueden interpretar como valores lógicos. APL proporciona un rico conjunto de funciones y algunas formas funcionales.

8.4.2.2 Funciones

Al contrario que LISP, APL proporciona un gran número de funciones (llamadas operaciones en la terminología APL). Una operación es monádica (toma un solo parámetro) o diádica (toma dos parámetros).

Todas las operaciones que son aplicables a escalares también

actúan sobre matrices. Así pues, $A \times B$ consiste en multiplicar A por B. Si A y B son ambos escalares, entonces el resultado es un escalar. Si ambos son matrices del mismo tamaño, el resultado es la multiplicación elemento por elemento. Si uno es una matriz y el otro es un escalar, se multiplica cada elemento de la matriz por el escalar. Cualquier otra combinación de tipos resulta indefinida.

Se proporcionan las operaciones aritméticas usuales +, -, ×, ÷, | (resto) y las operaciones lógicas y de relación usuales , ≤, =, ≥, >, ≠. Pero hay también operaciones no usuales.

La operación ";" es un "generador" que se puede usar para producir un vector de enteros. Por ejemplo, ;5 produce

1 2 3 4 5

La operación ";" concatena dos matrices. Así ;4; ;5 resulta

1 2 3 4 1 2 3 4 5

La operación " ρ " transforma su operando derecho en una matriz de las dimensiones deseadas, dadas en el operando izquierdo. Ejemplo:

2 2 ρ 1 2 3 4 - 1 2
3 4

2 3 ρ 1 2 3 4 5 6 - 1 2 3
4 5 6

La operación de compresión "/" toma dos argumentos de la misma dimensión y selecciona elementos de su argumento derecho, dependiendo de si el correspondiente elemento del argumento izquierdo es 1 o 0 en valores lógicos. Ejemplo:

1 0 0 1 / ;4 - 1'4

El argumento izquierdo puede consistir en expresiones lógicas. Por ejemplo:

A<B B<C C<D / X

extraerá ciertos valores de X, dependiendo de las comparaciones a la izquierda. En este caso X debe ser un vector de 3 elementos.

Las líneas de una función definida por el usuario se numeran consecutivamente comenzando por 1. Una línea puede estar etiquetada, en cuyo caso la etiqueta es igual al número de la línea. La única estructura de control de APL es la bifurcación, representada por "-->": ésta transfiere control a la línea cuyo número se especifica como argumento. Por ejemplo, -->3 transfiere control a la línea 3 de la función. Si el operando es 0 o mayor que el número de líneas de la función, la bifurcación se define

como una función de retorno; si el operando es un vector, se usa realmente el primer elemento, y si el operando es nulo no se realiza ninguna bifurcación.

La operación de bifurcación puede combinarse con la operación de compresión para construir bifurcaciones condicionales y múltiples. Por ejemplo, -->(A>B A=B A<B / caso1 caso2 caso3) transfiere control a la etiqueta apropiada dependiendo de los valores relativos de A y B.

-->(A<B / caso1)

bifurca a caso1 si A<B; en caso contrario, se ejecuta la sentencia siguiente.

En este momento, debe estar claro para el lector que la bifurcación no es una operación apropiada para un lenguaje funcional. Una de las razones por las que APL es un lenguaje no funcional es que no permite la construcción jerárquica de expresiones. Otra característica no funcional o imperativa es la asignación (←), que asigna el valor del operando derecho a la variable en el operando izquierdo. La sentencia de asignación produce un valor, que además puede usarse para construir expresiones; por ejemplo,

D ← C+Bx(A←2)

Sin embargo, la utilización de asignaciones en las expresiones introduce la posibilidad de efectos laterales.

Hay muchas otras operaciones primitivas en APL. Estas operaciones se pueden considerar como funciones matemáticas, ya que actúan sobre operandos y producen valores. Las funciones definidas por el usuario son similares a las funciones primitivas y también pueden ser monádicas o diádicas (las fuciones niládicas se corresponden con los procedimientos). Se usan con notación infix, y por lo tanto se pueden usar en expresiones al igual que las funciones predefinidas.

6.4.2.3 Formas Funcionales

Hay tres formas funcionales (operadores en la terminología de APL) proporcionadas por APL. Estos operadores actúan sobre las operaciones primitivas de APL y producen otras operaciones. Son:

- a. El operador reducción "/" (el mismo símbolo que compresión), es equivalente a la "inserción" de los lenguajes PF (también con el mismo símbolo). Por ejemplo, la suma de los elementos del vector A viene dada por

+/A

Una vez más, comparamos esto con la suma de los

elementos de un vector en lenguajes de programación imperativos. La repetición y el cálculo paso por paso se maneja a través de la forma funcional.

Si el operando derecho es una matriz, la operación de reducción se aplica sucesivamente a cada línea, es decir, si A es la matriz

1 2
3 4

entonces $+/A$ es

3
7

que se representa como

3 7

En general, una reducción aplicada a una matriz de n dimensiones produce una matriz de $n-1$ dimensiones.

- b: El operador producto escalar " $\cdot.$ " toma dos operaciones diádicas como argumentos y produce una operación diádica como resultado. Los operandos de una operación formada de esta manera deben ser matrices con tamaño "compatible". Por ejemplo, si son matrices, el número de filas del operando izquierdo debe ser el mismo que el número de columnas del operando derecho; el resultado será una matriz con tantas filas como el operando izquierdo y tantas columnas como el operando derecho. Si f y g son dos funciones diádicas primitivas, el efecto de

$Af.gB$

es aplicar g, elemento por elemento, a las filas correspondientes de A y a las columnas de B (p. ej.: primera fila de A con primera columna de B, etc). Todo ello seguido de la reducción de f ($f/$) sobre el vector resultante.

Como ejemplo de la potencia del producto escalar en la construcción de operaciones, digamos que la multiplicación de matrices se puede realizar mediante

$+.x$

Una vez más, podemos ver la potencia de las formas funcionales comparando esta solución con un procedimiento de multiplicación de matrices en un lenguaje como ALGOL, por ejemplo.

- c. El producto externo " $\cdot.$ " toma una operación primitiva

como operando y produce una operación diádica como resultado. La operación $.f$ aplicada a las matrices A y B (p. ej.: $A.fB$) tiene el efecto de aplicar f a cada elemento de A con todos los elementos de B. Por ejemplo, si A tiene el valor (1 2 3) y B tiene el valor (5 6 7 8), el resultado de $A .xB$ es la matriz

5	6	7	8
10	12	14	16
15	18	21	24

El efecto puede verse como la formación de una matriz con las filas etiquetadas con los elementos de A y las columnas etiquetadas con los elementos de B. Los elementos de la matriz son el resultado de aplicar la operación sobre las etiquetas de la fila y de la columna de cada elemento.

x	5	6	7	8
-----+-----+-----+-----+-----				
1	5	6	7	8
2	10	12	14	16
3	15	18	21	24

El producto externo tiene muchas aplicaciones en el proceso de datos cuando se crean tablas de porcentajes de interés, de impuestos, etc. También tiene otros usos, por ejemplo, para encontrar qué elementos de A aparecen en B.

$A \cdot.=B$

proporciona un mapa de valores lógicos, con un 1 en la posición donde el elemento de A es igual a un elemento de B.

Además de estas tres formas funcionales, hay dos operaciones en APL que se pueden usar para modificar el comportamiento de algunas otras operaciones. Estas son el especificador de eje y el operador recorrer. El especificador de eje "[]" nos permite especificar junto con la dimensión del operando, la operación que se aplica, diciendo qué índice variará más rápidamente. Por ejemplo, si la matriz A tiene un valor

1	2	3
4	5	6

la operación de reducción $+/A$, dará el resultado

6
15

Es decir, los elementos que se suman tienen los índices (1,1),

$(1,2), (1,3)$ y $(2,1), (2,2), (2,3)$. El índice que varía más rápidamente es el segundo. Por lo tanto, $+/A$ es equivalente a $+/[2]A$. Si en vez de esto quisieramos sumar las columnas de A , escribiríamos

$+/[1]A$

que dará el resultado

5 7 9

El operador recorrer " \backslash " aplica la operación de su primer argumento sucesivamente a lo largo de la secuencia de su otro argumento. Es útil para producir totales progresivos y en operaciones similares. Por ejemplo,

$+/\backslash 1 2 3 4$

produce

1 3 6 10

A pesar de sus muchas características funcionales, APL no consigue suficiente independencia de la máquina. La operación de asignación está presente todavía, y los efectos laterales son un medio común de obtener resultados. Hay muy pocas formas funcionales y sólo pueden aplicarse a funciones primitivas. Esta es una deficiencia importante para un lenguaje funcional, ya que significa que la potencia de las formas funcionales no se puede explotar en profundidad.

8.4.2.4 Un programa en APL

Volvamos ahora al problema de producción de números primos comprendidos entre 1 y N . La escueta información dada en esta sección sobre APL es, no obstante, suficiente para escribir este programa. El propósito de esta sección es ilustrar el estilo de programación APL en contraste con el de Pascal, por ejemplo.

No podemos usar una solución directamente basada en la estrategia que se deriva de la Sección 8.2.2, ya que APL no tiene la forma funcional "aplicar a todo". Incluso si la tuviera, no está permitido aplicar las formas funcionales a funciones definidas por el usuario.

Nuestro énfasis deberá estar en las asignaciones y repeticiones de matrices y expresiones en lugar de en las de escalares. Pensando en APL, debemos obtener un vector de números primos. Podemos comenzar con un vector de los números desde 1 hasta N y comprimirlo utilizando el operador compresión hasta que sólo contenga los números primos. En otras palabras, nuestra tarea es encontrar el vector de expresiones lógicas en el siguiente programa APL:

vector de expresiones lógicas/ N

Podemos partir de la definición de número primo: un número que es divisible solamente por 1 y por sí mismo. Así pues, para cada número en el rango que nos interesa, de 1 a N , podemos (a) dividirlo por todos los números del rango y (b) detectar aquellos que son divisibles por dos números.

El paso (a) puede realizarse con la operación resto y el producto externo

$(\cdot N) \cdot . | (N)$

El resultado de esta operación será un vector de restos. Nosotros estamos interesados en comprobar si el resto es igual a 0.

$0=(\cdot N) \cdot . | (\cdot N)$

Ahora el resultado es una matriz lógica indicando si los números son divisibles (1) o no lo son (0).

En el paso (b), queremos ver cuántas veces es divisible cada número, es decir, el número de unos en cada fila.

$+/[2]0=(\cdot N) \cdot . | (\cdot N)$

Pero sólo estamos interesados en aquellas filas que tienen exactamente dos unos.

$2=(+/[2]0=(\cdot N) \cdot . | (\cdot N))$

El resultado es un vector lógico indicando si el índice es primo (1) o no lo es (0). Este es el vector de expresiones lógicas que buscábamos. Para conseguir los números primos aplicamos la compresión

$(2=(+/[2]0=(\cdot N) \cdot . | (\cdot N))) / : N$

La esencia de esta solución es la combinación de expresiones sucesivamente más complejas, a partir de otras más simples. Siendo éste el único mecanismo que se usa para combinar acciones. Este mecanismo de combinación es simple y uniforme independientemente de los componentes que se combinan. Es matemático, en el sentido en que todas las operaciones y expresiones se usan de la misma forma que se usan en las matemáticas.

Nuestro programa es un típico APL unilineal. Se ha dicho que APL tiene como defecto que sus programas unilinea no son fáciles de leer. Pero, para ser justos, la legibilidad del anterior programa unilineal se debe comparar con la legibilidad del programa Pascal completo que realice la misma tarea, en vez de con una sentencia Pascal. Así, un programador de APL entiende el programa APL sin más esfuerzo que el requerido por un programador

de Pascal para entender un programa Pascal. Una consideración importante es que los programadores deben tener soltura con el lenguaje y su estilo.

8.5 COMPARACION ENTRE LENGUAJES FUNCIONALES E IMPERATIVOS

Vamos a revisar las diferencias entre los lenguajes de programación funcionales y los imperativos. Los lenguajes imperativos se basan en los ordenadores convencionales, mientras que los funcionales se basan en las funciones matemáticas. Las características de las dos clases de lenguajes están marcadas por sus diferentes fundamentos.

Los lenguajes imperativos son más eficientes en términos de tiempo de ejecución, ya que reflejan la estructura y operaciones de la máquina. Sin embargo, debido a esto requieren que el programador ponga atención en detalles a nivel de la máquina. Esta influencia puede verse en el estilo de programación suscitado por estos lenguajes, el cual se basa en la denominación de las celdas elementales, asignación de valores a estas celdas, y repetición de acciones elementales.

El estilo de la programación funcional no depende de estas tres acciones, ya que la simplicidad y uniformidad de los objetos de datos (p. ej., secuencias, listas, matrices) permiten el diseño de estructuras de datos sin preocuparse por las celdas de memoria. En vez de asignarse, los valores se producen por la aplicación de una función y se pasan como argumentos a otras funciones; y las formas funcionales y operaciones que actúan sobre los objetos (como en APL) reducen la dependencia de la repetición. En conjunto, la programación funcional parece ser de más alto nivel que la programación imperativa. Así pues, podría hacer la programación más sencilla.

El coste de esta facilidad de programación se manifiesta en términos de eficacia de ejecución. La ineficacia procede no sólo de todas las llamadas a funciones, sino también del hecho de que muchos objetos se crean y se destruyen dinámicamente. La creación dinámica de objetos tales como listas y matrices no se puede soportar eficazmente (en LISP, se inventó el "recolector de residuos" para tratar este problema, ver Sección 4.7.4).

Hemos visto que la eficacia es tan importante que APL y LISP han optado por características no funcionales con el fin de conseguirla. Pero debemos darnos cuenta que las consideraciones de eficacia podrían ser diferentes con un modelo de máquina diferente.

Por ejemplo, consideremos el programa de números primos de la Sección 8.1.1. Una técnica para aumentar la velocidad de nuestro programa Pascal es tener en cuenta los números primos que han sido ya calculados antes de la generación de un nuevo número primo. En concreto, sólo necesitamos probar si un número es

divisible por números primos. Sin embargo, en APL y LISP la tendencia es repetir exactamente la misma operación para todo el conjunto. No obstante, si el programa se ejecutara en un multiprocesador en el cual cada número primo se calcula en un procesador separado, entonces los programas APL o LISP podrían ser incluso más eficientes.

Un lenguaje de programación funcional proporciona una forma natural de explotar una arquitectura de máquina paralela, ya que las aplicaciones internas pueden ser aplicadas en paralelo.

Estos ejemplos muestran el peligro de diseñar nuestro lenguaje de programación en términos de eficiencia. La eficiencia está dictada por la máquina, y la máquina debería estarlo para soportar el lenguaje, por lo que quedamos atrapados en un círculo vicioso. Idealizando, deberíamos diseñar un lenguaje basado en lo que conocemos como buenas estrategias de resolución de problemas, y después podremos diseñar una máquina para soportar el lenguaje. Encontrar estrategias de resolución de problemas es nuestro primer desafío. Sin embargo, el diseño de una arquitectura para soportar el lenguaje es también de decisiva importancia. Si tenemos que continuar con nuestras máquinas tradicionales, las consideraciones de eficacia regularán los lenguajes funcionales. Actualmente se están investigando diferentes arquitecturas que soportan lenguajes funcionales, pero es prematuro establecer si llegarán a ser relevantes en la práctica.

A parte de los problemas de eficacia, es difícil determinar si los lenguajes puramente funcionales tales como PF serán satisfactorios. El éxito de un lenguaje no depende tanto de la elegancia matemática de sus principios elementales como de una combinación compleja de sus características externas, los sistemas que lo soportan y otras herramientas disponibles en él.

Hay un largo camino por recorrer antes de que los PF produzcan un lenguaje utilizable. ¿Necesitamos los tipos? ¿Entrada/salida? ¿Qué funciones primitivas y formas funcionales? La dirección tomada en estos objetivos prácticos determinará en gran medida si el lenguaje será aceptado en programación.

Otra razón de la dificultad en la valoración de los PF es que en realidad, todavía no tenemos experiencia con ellos. Podemos criticar mucho más fácilmente los lenguajes convencionales porque hay muchos programadores usándolos y rechazándolos, y así se obtienen datos sobre la debilidad de estos lenguajes. Incluso aunque es una hipótesis razonable, no hay una evidencia empírica de que la descomposición funcional de problemas haga más sencilla la programación.

La prueba final para cualquier lenguaje será cuando se esté usando por un gran número de programadores en tareas de programación cotidianas.

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

Los lenguajes PF aparecen en (Backus 1978), que junto con (Backus 1973) son la principal fuente de información para este capítulo. Estos artículos han motivado un gran interés en la programación funcional y sus aplicaciones.

(Pozefsky 1977) discute la programación en el lenguaje de Backus. (Berkling 1980) y (Magó 1980) presentan una arquitectura de máquina que soporta la ejecución eficaz de lenguajes de programación funcionales.

(Henderson 1980) es una excelente introducción a la programación funcional que también cubre la implementación de un lenguaje funcional en un ordenador convencional.

Tanto APL como LISP se utilizan profusamente y son muy populares entre sus usuarios. Ambos lenguajes están soportados por entornos completos. La discusión de estos lenguajes en este capítulo ha sido sólo un análisis superficial. Los fundamentos de LISP se desarrollan en (McCarthy 1960). (McCarthy y otros 1965), (Siklóssy 1976) y (Allen 1978) proporcionan buenas descripciones de LISP. (Sandewall 1978) y (Teitelman y Masinter 1981) son estudios de sistemas LISP productivos y agradables de usar. (Steele y Sussman 1980) describe un microprocesador diseñado para ejecutar LISP. APL se definió en el libro A Programming Language (Iverson 1952), del cual se deriva su propio nombre. (Polivka y Pakin 1975) proporciona una visión clara de APL. (Iverson 1979) discute la utilidad de las formas funcionales en APL.

Otras bases para el diseño de lenguajes no mencionadas en este capítulo han sido tratadas en: SETL (Schwartz 1973), (Kennedy y Schwartz 1975) que se basan en la teoría de conjuntos; LUCID (Ashcroft y Wadge 1977) que se basa en la lógica; SNOBOL4 (Griswold y otros 1971) que se basa en los algoritmos de Markov.

Ejercicios

8.1 La función $\alpha+$ de la Sección 8.3.2.e requiere que cada secuencia interna a su secuencia argumento sea de longitud 2. Modifique esta función para que sea capaz de trabajar con secuencias internas de cualquier longitud.

8.2 Usando PF, escriba la función "COMBINAR" definida como

COMBINAR: $\langle\langle x_1, x_2, \dots, x_n \rangle, \langle y_1, y_2, \dots, y_n \rangle\rangle =$
 $\langle\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots, \langle x_n, y_n \rangle\rangle$

Es decir, "COMBINAR" se aplica sobre dos secuencias de igual longitud y produce una secuencia compuesta de secuencias de longitud 2. La i-ésima secuencia interna en la secuencia resultante consta del i-ésimo elemento del primer argumento y del i-ésimo elemento del segundo argumento. Utilice

cualquier función primitiva de este capítulo que necesite.

8.3 Escriba una función que sume dos vectores representados como secuencias:

SUMARV: $\langle\langle x_1, x_2, \dots, x_n \rangle, \langle y_1, y_2, \dots, y_n \rangle\rangle =$
 $\langle x_1 + y_1, \dots, x_n + y_n \rangle$

8.4 Escriba una función que sume cualquier número de vectores:

SUMARVS: $\langle\langle x_{11}, x_{12}, \dots, x_{1n} \rangle, \langle x_{21}, x_{22}, \dots, x_{2n} \rangle, \dots, \langle x_{m1},$
 $\dots, x_{mn} \rangle\rangle = \langle x_{11} + x_{21} + \dots + x_{m1}, \dots, x_{1n} + x_{2n} + \dots + x_{mn} \rangle$

8.5 Escriba el programa de números primos de la Sección 8.2.2 en PF.

8.6 Defina la forma funcional "REPETIR".

8.7 Explique por qué el COND de LISP es una función y no una forma funcional.

8.8 Diseñe un lenguaje de programación conveniente para operaciones con matrices tales como suma y multiplicación de matrices. ?Qué funciones primitivas y qué formas funcionales son necesarias y/o útiles?

8.9 Los PF tal y como los definió Backus tienen un importante atributo que se ha tratado en este capítulo. Tienen un valor especial "i" llamado "bottom" o "indefinido", el cual se devuelve por las funciones en el caso de condiciones anormales. Por ejemplo, si una función se aplica sobre argumentos no esperados, o si uno de los argumentos es "i", se devolverá "i". ?Porqué es ésta una propiedad útil? ?Cómo se debería modificar la definición de la función primitiva en la Sección 8.3.1 para recoger esta característica?

8.10 Escriba el programa de números primos usando únicamente características funcionales de LISP.

DISEÑO
DEL
LENGUAJE

"... consolidación, no innovación ..." (Hoare 1973)

La razón de ser de este libro estriba en el hecho de que los lenguajes de programación son herramientas de producción de software. Los capítulos anteriores han expuesto en profundidad este punto de vista, tratando conceptos de lenguajes de programación y comparando y evaluando las múltiples soluciones adoptadas por los lenguajes existentes. En concreto, hemos sugerido unos cuantos criterios para la evaluación de estos lenguajes, centrados alrededor de los conceptos de tipos de datos, estructuras de control, corrección, y programación a gran escala. No obstante, desde una perspectiva ligeramente diferente, estos criterios también sugieren unas directrices para el diseño de lenguajes de programación. Esta es la razón por la cual muchas de las consideraciones que veremos, ya han sido mencionadas en capítulos anteriores. El objetivo de este capítulo es principalmente recapitulatorio, por lo tanto intentaremos reunir las múltiples facetas del problema y, en varios casos, mostraremos cómo diferentes características del lenguaje, cada una deseable por sí sola, puede a menudo interferir con otra cuando se combinan.

Según C.A.R. Hoare (Hoare 1973), "El diseñador del lenguaje debería estar familiarizado con muchas características alternativas diseñadas por otros, y tener un buen criterio para elegir la mejor y rechazar las que sean incompatibles. Debe ser capaz de solucionar mediante un buen diseño, cualquier pequeña incompatibilidad o solapamientos entre características diseñadas por separado. Debe tener una idea clara del ámbito y propósito de la aplicación de su nuevo lenguaje, y cómo deberá ser en cuanto a tamaño y complejidad. . . Lo que no deberá hacer es incluir ideas propias no experimentadas. Su tarea es la consolidación, no la innovación".

Al diseñar un lenguaje, también es necesario diseñar una implementación para él. Wirth (Wirth 1975a) dice, "En la práctica, un lenguaje de programación es tan bueno como sus compiladores." De hecho, mucha de la popularidad de los viejos lenguajes, tales como FORTRAN, probablemente proviene de la disponibilidad de compiladores fiables y eficientes, es decir, compiladores que producen código eficiente y tienen buenos sistemas de diagnóstico. Wirth asegura que "un buen lenguaje debe formarse a partir de ideas claras de objetivos de diseño junto con intentos simultáneos para definirlo en uno, o, preferiblemente, en varios ordenadores." Estos temas serán tratados más adelante, especialmente en la Sección 9.2.

Por último, no se debería olvidar que el usuario de un lenguaje a menudo interactúa con otros instrumentos provistos por el sistema de programación. En la mayoría de las soluciones existentes, como ya hemos visto, tales interacciones no se llevan a cabo en un sistema de programación único, sino que se utilizan herramientas de soporte independientes, tales como editores de

texto, montadores y compiladores. Esto a menudo permite al programador burlar las características del lenguaje para aumentar la fiabilidad del programa. Por ejemplo, los comandos del lenguaje de control de trabajos se pueden usar para combinar varias unidades compiladas separadamente, en un único programa ejecutable. Sin embargo, el montador provisto por el sistema normalmente no comprueba la comunicación entre los componentes. De igual forma, el editor de textos estándar suministrado por el sistema, no garantiza que un programa correcto se deje en un estado correcto después de una modificación. La integración del lenguaje con las herramientas en un único sistema coherente de programación, debería ser un objetivo prioritario de diseño, y de hecho, hay una tendencia muy clara en esta dirección.

9.1 CRITERIOS DE DISEÑO

Los lenguajes de programación deben ayudar al programador en el diseño, documentación y validación de programas. Por lo tanto, los criterios para el diseño de un lenguaje pueden clasificarse de acuerdo a los siguientes principios: facilidad de escritura, legibilidad y fiabilidad. Cada uno de estos principios será discutido por separado en las Secciones 9.1.1, 9.1.2 y 9.1.3.

También deberíamos recordar que, al fin y al cabo, los lenguajes de programación son ejecutados por ordenadores. Traducción y ejecución eficiente son, por lo tanto, dos objetivos adicionales en el diseño del lenguaje. En la mayoría de los casos prácticos, la ejecución eficiente del código compilado es más importante que la traducción eficiente, ya que los programas, una vez entregados, se ejecutan muchas veces, o incluso de forma continua (en el caso de programas monitores) sin volver a ser compilados. Sin embargo, en ciertos sistemas (por ejemplo, en educación e investigación), los programas se compilan casi cada vez que se van a ejecutar.

Los requisitos del traductor han influido algunas veces en el diseño del lenguaje, pero solamente en aspectos de poca importancia. Un ejemplo es la regla usual (p. ej., en Pascal) de que cada nombre debe ser declarado antes de usarlo, lo que permite que los programas se traduzcan en un solo paso. (Esta regla, no obstante, también tiene una justificación desde el punto de vista de la legibilidad).

La ejecución eficiente es un requisito difícil de conseguir si no se tiene en cuenta la arquitectura de una máquina en particular. Se asumirá como objetivo del diseño en la siguiente sección, pero bastante informalmente. Merece la pena mencionar que en algunos casos, la ejecución eficiente se convierte en un objetivo de diseño tan importante que el lenguaje refleja directamente algunos aspectos de la arquitectura subyacente. Esto es especialmente cierto en los lenguajes para programación de sistemas. Más adelante se discuten los criterios de diseño para lenguajes independientes de la máquina, es decir, lenguajes previstos para ejecutarse en cualquier máquina. En la sección de

Ampliación Bibliográfica se dan referencias de lenguajes de alto nivel dependientes de la máquina.

La independencia de la máquina es un objetivo del diseño para cuyo logro se requiere una especial atención. Muchos lenguajes supuestamente independientes de la máquina resultan tener características dependientes de ésta. Un ejemplo es la restricción en la cardinalidad de conjuntos impuestos por ciertas implementaciones para hacer que dichos conjuntos sean representables por una única palabra.

9.1.1 Facilidad de Escritura

Los lenguajes de programación ayudan en el diseño del programa proporcionando construcciones que hacen que el método de diseño adoptado sea fácilmente expresable. Esto implica que el programador puede concentrarse en la comprensión y resolución del problema en lugar de preocuparse de las herramientas usadas para expresar las soluciones. Las propiedades básicas que contribuyen a la facilidad de escritura se pueden reducir a cuatro: simplicidad, expresividad, ortogonalidad y definición. Estas propiedades son difíciles de definir, y a menudo se complementan unas a otras.

9.1.1.1 Simplicidad

Un lenguaje debería ser fácil de conocer. Todas las diferentes características deberían ser fáciles de aprender y recordar, y el efecto de cualquier combinación de ellas, predecible y de fácil comprensión. Un ejemplo típico de un lenguaje que no satisface este requisito es PL/I. La gran cantidad de diferentes características que están presentes en el lenguaje lo hacen difícil de aprender en su totalidad; incluso una sola ojeada al tamaño del manual de referencia puede ser desalentadora y frustrante.

No obstante, se dice a menudo que el programador que no puede (y no necesita) comprender el lenguaje en su integridad, puede vivir felizmente con un subconjunto. Tal y como Hoare (Hoare 1973) señala, esta pretensión no está justificada. El conocimiento de un subconjunto puede ser suficiente para que los programas hagan lo que el programador pensó, pero si el programa no trabaja apropiadamente, y accidentalmente se llama a alguna función desconocida del lenguaje, entonces se le pueden presentar serios problemas al programador.

La legibilidad se ve deteriorada si el lenguaje proporciona varias maneras alternativas para la especificación del mismo concepto. Dando más de una forma para denotar un mismo concepto, se incrementa el tamaño del lenguaje y se favorece el desarrollo de "dialectos" que usan solamente subconjuntos de tales formas. Un programador experto en un dialecto puede tener dificultades para la comprensión de programas escritos en un dialecto

diferente. Un ejemplo de este problema se puede encontrar en COBOL, el cual soporta dos notaciones, una concisa y matemática, y otra de tipo inglés. Por ejemplo, se puede escribir

MULTIPLY HORAS-TRABAJO BY PAGA-HORA GIVING PAGA-DIARIA.

o

PAGA-DIARIA = HORAS-TRABAJO * PAGA-HORA.

PL/I ofrece muchas características que causan el mismo problema. Por ejemplo: 1) El acceso a los componentes de un registro se puede hacer por calificación completa (especificando todos los campos selectores) o por calificación parcial (cuando no hay ambigüedad). 2) Las declaraciones tienen valores por defecto para atributos no especificados.

C ofrece muchos ejemplos de este problema. La suma de 1 a la variable entera *a* se puede efectuar con cualquiera de las siguientes cuatro sentencias:

- 1) *a++;*
- 2) *a = a+1;*
- 3) *a += 1;*
- 4) *++a;*

Al elemento *i* de una matriz *a* se puede acceder por *a[i]*, o por **(a+i)*.

A un componente *x* de una estructura *S* apuntada por un puntero *p* se puede acceder por *p->x* o por *(*p).x*.

Ada sufre problemas similares. Por ejemplo, los parámetros se pueden transmitir a los subprogramas por el método del nombre, o por el método posicional, o con una mezcla de ambos (Sección 5.2.1). Los valores de un registro se pueden especificar de una forma similar. Por ejemplo, a una variable *X* del tipo *T* descrito a continuación

```
type T is
  record
    A: CHARACTER;
    B: INTEGER;
  end record;
```

se le puede asignar un valor con la siguiente sentencia

X := (B-->3, A->'F');

o por medio de

X := ('F',3);

La simplicidad también se ve disminuida si el lenguaje permite que diferentes conceptos se expresen con la misma

notación. La sobrecarga es un ejemplo. Ya hemos dicho que un uso juicioso de la sobrecarga puede ser útil (Sección 4.1). Normalmente, los operadores aritméticos están sobre cargados, lo cual hace el lenguaje más simple. Sin embargo, algunos lenguajes (p.ej., ALGOL 68 y Ada) generalizan el concepto de sobrecarga a operadores y subprogramas definidos por el usuario. Una llamada a un subprograma sobre cargado es ambigua (e ilegal) si los tipos y el orden de los parámetros reales (o los nombres de los parámetros formales en el método de paso de parámetros por nombre en Ada) no son suficientes para identificar exactamente una declaración de subprograma. Con la sobrecarga y las reglas de ámbito de un lenguaje, se puede conseguir fácilmente que un programa presente dificultades en su lectura.

Por último, existen casos en los cuales, características diferentes, y aparentemente simples, del lenguaje interactúan de una forma crítica e producen programas cuyo comportamiento es difícil de predecir. Por ejemplo, considerar la interacción entre el paso de parámetros a subprogramas y el manejo de excepciones en Ada. Un subprograma que provoca una excepción puede producir resultados diferentes dependiendo de si el paso de parámetros está implementado por referencia o por copia. Un ejemplo de la combinación de características que generan un comportamiento difícil de predecir, se ilustra con el manejo de excepciones y multitareas en Ada. (Animamos al lector a que se dirija al manual del lenguaje en lo que se refiere a este punto).

Sin embargo, la simplicidad por sí sola puede ser un objetivo decepcionante. Por ejemplo, los lenguajes máquina son, usualmente, simples en el sentido de que el número de instrucciones máquina es a menudo pequeño, y cada instrucción tiene un efecto definido y de fácil comprensión. No obstante, la dificultad de la programación en lenguaje máquina no estriba estrictamente en el uso del lenguaje, sino en su bajo nivel. La complejidad no está en el lenguaje, sino en la dificultad para representar la solución con acciones elementales ejecutadas por la máquina sobre unos datos sin estructura.

3.1.1.2 Expresividad

La discusión anterior critica los lenguajes máquina debido a su bajo poder expresivo. La expresividad de un lenguaje es una medida de la naturalidad con que se puede representar la estrategia de la solución de un problema mediante una estructura de un programa. Un punto principal expuesto en este libro es que la abstracción en los datos y en el control, junto con los mecanismos de modularización, son las herramientas básicas de estructuración en el diseño de un programa. Los puede usar el programador a varios niveles, desde la estrategia general, en la descomposición de una tarea compleja en subtareas más simples y en el diseño de los interfaces entre las subtareas, hasta los detalles de la codificación y la representación de los datos.

Pascal es quizás el ejemplo más conocido de lenguaje simple

con un alto poder expresivo, a pesar de su falta de mecanismos adecuados para la modularización. Su rico conjunto de estructuras de control y tipos, es responsable de gran parte de su éxito.

9.1.1.3 - Ortogonalidad

La atractiva simplicidad y expresividad de Pascal se ve a menudo deteriorada por su carencia de ortogonalidad. Ortogonalidad significa que se debería permitir cualquier composición de primitivas básicas. Esto conseguiría el mayor grado de generalidad, sin ninguna restricción o casos especiales. Un ejemplo típico de la carencia de ortogonalidad en Pascal está en los procedimientos (funciones) y las llamadas a éstos, los cuales están sujetos a las siguientes restricciones:

- Los ficheros no se pueden pasar por valor.
- Los componentes de una estructura de datos empaquetados no se pueden pasar por referencia.
- Los procedimientos y funciones pasados como parámetros, solamente pueden tener parámetros por valor.
- El tipo de los parámetros formales debe ser establecido en el encabezamiento de los procedimientos o funciones, excepto si los parámetros son procedimientos.
- Las funciones sólo pueden devolver valores de un conjunto de tipos restringido; no pueden devolver matrices, registros, conjuntos ni ficheros.
- El tipo de los parámetros formales solamente se puede especificar por un identificador de tipo, y no por su representación.

Por ejemplo:

```
procedure noortogonal (var x:array [1..10] of real;
                      y: char)
no es legal, mientras que sí lo es
procedure noortogonal (var x:T; y:char)
donde T está declarado en un ámbito más externo como
type T = array [1..10] of real
```

Otro ejemplo está en las declaraciones de constantes en Pascal, tales como `const nulo = 0,` las cuales permiten que las constantes tengan un nombre simbólico. Pascal no permite la asignación de un valor constante a punteros u objetos estructurados (tales como matrices y registros). Finalmente, las variables de un tipo enumerado sólo pueden inicializarse por

medio de asignaciones, y no por valores de entrada; de hecho, tales variables no se pueden leer (ni escribir). Esto, a menudo, hace que los programas Pascal tengan soluciones inadecuadas. El programador está forzado a codificar los valores de entrada de una variable de tipo enumerado, por ejemplo en enteros, y después convertir los valores leídos por medio de sentencias del programa.

La carencia de ortogonalidad puede ser molesta. El programador no puede aplicar las generalizaciones de una manera uniforme, porque podría ser ilegal, y con frecuencia debe dirigirse al manual del lenguaje para confirmar la legalidad y corrección de lo que está escribiendo.

ALGOL 68 es quizás, el mejor ejemplo de un lenguaje de programación ortogonal. Por ejemplo, se permiten objetos de cualquier tipo como parámetros y resultados de subprogramas. En cuanto a las declaraciones de identidad, los mecanismos del ALGOL 68 para la definición de constantes, tienen la forma

```
m.id = e
```

donde m es un modo, id es un identificador, y e es una expresión. Aquí, m puede ser cualquier modo (permitido) y e puede ser cualquier expresión cuyo valor se asigna sin modificación a id. Incluso más generalmente, las declaraciones de identidad se utilizan para declarar procedimientos y funciones en el modo convencional, tal como en

```
proc sum = (real a, b) real: a+b;
```

Como objetivo de diseño, la ortogonalidad no se deberá considerar como una sustitución de la simplicidad. Sin embargo, el compromiso entre los dos es difícil de cuantificar y viene a ser una decisión fundamental en el diseño del lenguaje. No está claro si es mejor tener unos pocos conceptos simples libres de restricción, o una colección más grande de conceptos menos simples con varias restricciones pequeñas aunque molestas. De hecho, la composición de características ortogonales en ALGOL 68 puede conducir a unos programas excesivamente difíciles de entender (y compiladores difíciles de implementar). Por ejemplo, sentencias convencionales, pueden en principio producir un valor, tal como se ilustra en el siguiente fragmento:

```
real x, y; read ((x, y)); if x<y then a else b fi) :=
b+if a := a+1; a>b then c := c+1; +b
else c := c-1; a
fi
```

La parte izquierda de esta sentencia de asignación es una unidad que declara las variables locales x e y, y produce la variable a o b como su valor, dependiendo de los valores leídos para x e y (se asume que a, b y c son variables de tipo real). El segundo

operando de la expresión de la parte derecha está expresado como un condicional. El valor de

`a := a+1; a>b`

es el valor del componente final, es decir, un booleano. De forma similar, el valor de

`c := c+1; +b`

es el valor real $+b$, que puede usarse como componente de una expresión.

Como dice Wirth (Wirth 1975a), "Si intentamos producir simplicidad por medio de la generalidad del lenguaje, podemos obtener programas que debido a su falta de concisión sobrepasan nuestro límite alcance intelectual...". Por esto, la clave no está en minimizar el número de características básicas del lenguaje, sino en mantener las características fáciles de comprender con todas sus consecuencias de uso, y libres de interacciones inesperadas cuando se combinan. Al equilibrio entre ortogonalidad y simplicidad que necesitamos conseguir, se le podría llamar "predictibilidad". La combinación de las características del lenguaje no debería producir resultados inesperados; y, basados en el conocimiento de las primitivas del lenguaje, debería ser posible predecir el efecto de sus combinaciones.

Podemos decir, en términos bastante simplistas, que Pascal es simple y bastante expresivo, pero no ortogonal. ALGOL 68 es ortogonal y bastante expresivo, pero no simple. Ada es expresivo, pero no es ni simple ni ortogonal (el análisis de la ausencia de ortogonalidad en Ada, se lo dejamos al lector como un ejercicio).

Los lenguajes funcionales tienen la potencia de conseguir a la vez simplicidad y ortogonalidad, debido a la restricción en la interacción entre elementos del lenguaje y a la simplicidad de los mecanismos que los combinan.

9.1.1.4 Definición

Un punto importante para la facilidad de uso de un lenguaje de programación, es la precisión en la descripción de su sintaxis y su semántica. La vaguedad es un enemigo del programador en dos aspectos importantes. Primero, el programador no confía completamente en el lenguaje, y cualquier intento para encontrar una respuesta precisa refiriéndose a la definición estándar resulta en mera frustración. Segundo, se pueden elegir implementaciones diferentes para resolver características planteadas de una forma ambigua, con desafortunadas consecuencias para la portabilidad del programa.

Pascal está definido por un documento bastante informal que es atraktivamente simple y de fácil lectura, pero deja sin

respuesta un número considerable de sutiles cuestiones. La sintaxis del lenguaje está definida por una simple gramática BNF de contexto libre, la cual no excluye un infinito número de programas incorrectos. Por ejemplo, la gramática no especifica que una expresión aritmética tal como $x+3.75$ no se pueda asignar a una variable booleana. Por esto, los programas correctos sintácticamente son programas legales Pascal solamente si satisfacen un conjunto de condiciones adicionales descritas por un documento (Report) en prosa inglesa, con todas las ambigüedades y omisiones que esto implica. También se ha dado una definición formal axiomática de Pascal. Sin embargo, esta omisión no trata los procedimientos en su completa generalidad, y falla también en las respuestas a cuestiones acerca de la compatibilidad de tipos que también se dejan sin respuesta en el Report.

ALGOL 68 está definido por un documento extremadamente preciso: el "Revised Report". La sintaxis del lenguaje se describe por una gramática W, la cual tiene en cuenta todos los aspectos del lenguaje sensitivos al contexto. La descripción semántica se da en una versión estilizada en un inglés que raramente es ambiguo. La definición del ALGOL 68 es completa. El Report define explícitamente todas las comprobaciones y procesos que se hacen en tiempo de compilación (incluyendo la compatibilidad de modos), así como todas las acciones que se realizan en tiempo de ejecución (incluyendo la señalización de errores en ejecución).

El "Report" de ALGOL 68 está considerado generalmente como de difícil lectura; de hecho, se reconoció pronto la necesidad de una descripción más aceptable, lo cual condujo a una introducción oficial e informal al lenguaje (Lindsey y Van der Meulen 1977). Sin embargo, la falta de legibilidad es posiblemente más una propiedad de la definición formal que del propio lenguaje. De hecho, la cuestión de cómo proporcionar descripciones formales, claras y simples de los lenguajes de programación, todavía es materia de investigación.

7.1.2 Legibilidad

La legibilidad es el principal factor de influencia en la facilidad de modificación y mantenimiento de un programa. Consecuentemente, tiene un impacto considerable en el coste final de la producción de software. La legibilidad está estrechamente relacionada con la facilidad de escritura: las características de un lenguaje que favorecen la escritura, normalmente favorecen también la legibilidad. Esto no es de extrañar, ya que cuando se escribe un programa, el programador debe volver a menudo a leer partes de él. La simplicidad y expresividad ayudan en gran medida al programador a escribir código autodocumentado, e incluso a desarrollar un estilo de escritura clara. En particular, las abstracciones en los datos y en el control, y la modularización gradual del programa proporcionan la principal ayuda para la comprensión gradual del programa hasta los niveles de mayor detalle. Por

esto, la distinción entre facilidad de lectura y de escritura es a menudo bastante arbitraria.

Un aspecto importante en la legibilidad es la documentación del programa. El propósito de la documentación es explicar al lector humano cómo trabaja un programa, de tal forma que posteriores modificaciones para afrontar nuevos requisitos, o para mantener el programa, se puedan efectuar con facilidad y sin errores. Hay un consenso muy extendido por el que "el hecho de que la documentación es algo que se debe añadir después de que el programa ha sido terminado, parece ser error en principio y antiproductivo en la práctica. En su lugar, la documentación debe considerarse parte integrante del proceso de diseño y codificación" (Hoare 1973)..

Los comentarios desempeñan un papel importante en la documentación del programa. En los lenguajes orientados a líneas (p.e. lenguajes cuyas sentencias ocupan una línea), un comentario empieza, o bien en una posición fija, o bien con una marca especial reservada para este propósito. Por otra parte, los lenguajes orientados a cadenas (de sentencias) (p.ej., lenguajes en los que el texto se ve como una cadena continua de caracteres) introducen unos delimitadores especiales, tales como "(" y ")" en Pascal, para encerrar comentarios que pueden aparecer en cualquier parte del programa.

Esto ha tenido una desafortunada y frecuente consecuencia. Cuando al programador se le olvida el delimitador de cierre, partes enteras de programa son tomadas como comentarios, y por lo tanto ignoradas por el compilador. Aún peor es el caso de ALGOL 68, que utiliza el mismo símbolo (comment, co, o #) como delimitador de apertura y cierre. El olvido de uno de ellos puede causar que un comentario sea tratado como texto del programa, y viceversa. Los lenguajes orientados a línea no tienen este problema, ya que el final de la línea es interpretado como un delimitador de cierre implícito. Lo convenido en Ada, un comentario se termina implícitamente al final de la línea, es una buena solución.

Los convenios léxicos del lenguaje tienen una influencia en la legibilidad. Las restricciones estrictas en la longitud de los identificadores, como en FORTRAN y C, pueden forzar al programador a dar nombres crípticos a las variables y procedimientos. La legibilidad se realza más todavía, si se pueden utilizar el carácter de subrayado "_" o un espacio en los identificadores. Por ejemplo, un identificador de un campo de un registro, nombre_esposa o nombre espesa es preferible a nombreespresa. Pascal prohíbe los caracteres de subrayado y los espacios dentro de los identificadores. Sin embargo, por motivos de legibilidad, hemos ignorado en nuestros ejemplos esta molesta restricción. Un buen compromiso, si el lenguaje permite letras mayúsculas y minúsculas, es empezar cada parte del nombre con una letra mayúscula, por ejemplo, NombreEsposa.

Otro factor clave de influencia en la legibilidad de un

programa, es la sintaxis del lenguaje. Los delimitadores explícitos (if ... fi, do ... od, case ... esac, etc.) son preferibles a los pares begin ... end requeridos por Pascal para agrupar sentencias, ya que los primeros indican claramente el propósito del grupo de sentencias. En los lenguajes orientados a cadena, la estructura sintáctica del programa se puede hacer evidente adoptando convenios apropiados para el decalaje (indentación). Aunque éste es un método efectivo para auto-dокументación, es difícil de llevar a la práctica. El programador a menudo empieza escribiendo un programa con buenas intenciones, pero tan pronto como el programa experimenta alguna modificación, los cuidadosos decalajes iniciales se pierden, y el programa toma una estructura chapucera y confusa. Volver a decalar el programa completo puede ser costoso y propenso a error si se efectúa manualmente. Sin embargo, algunos sistemas de programación, contienen una herramienta especializada (llamada Pretty-printer) que permite volver a perfilar automáticamente el programa por medio de un procedimiento de proceso de texto.

La legibilidad del programa también se ve afectada por la semántica del lenguaje. En el Capítulo 6 tratamos ampliamente las características del lenguaje que hacen los programas difíciles de analizar, y también vimos propuestas que tienden a minimizar (o eliminar por completo) algunos efectos laterales no deseados. Por ejemplo, los procedimientos con parámetros pasados por referencia y las variables globales pueden producir alias. Euclid impone restricciones semánticas que hacen que el alias sea ilegal (y por lo tanto detectable).

Como ya hemos dicho, las restricciones semánticas hacen a los programas más legibles, al precio de una generalidad reducida. Por ejemplo, un sistema de bases de datos está basado intrínsecamente en el hecho de que distintos usuarios ven un conjunto de datos concurrentemente como una entidad lógica diferente para cada uno de ellos. Una base de datos particular podría ser vista por un programa de nóminas, como un conjunto de datos acerca de empleados individuales, mientras que desde un punto de vista de gestión, la misma base de datos se podría contemplar como un conjunto de grupos de trabajo organizados jerárquicamente. En otras palabras, los sistemas de bases de datos están basados intrínsecamente en efectos laterales y alias, y no hay esperanza de diseñar tales sistemas con un lenguaje que se deshaga de estas características.

Es importante apuntar, que en todo el examen previo acerca de la legibilidad, el término significa "legibilidad por una persona familiarizada con el lenguaje". Así, el hecho de que la sentencia Pascal

A := B+C

se pueda escribir en COBOL como

ADD B TO C GIVING A

no hace que COBOL sea más legible. La sentencia COBOL tiene una sintaxis tipo inglés, lo cual puede ocasionar que alguien que no sea programador tenga la falsa sensación de comprender el programa. Si realmente pudiéramos describir el programa entero en pseudo-inglés, ese lenguaje sería más legible para todas las personas familiarizadas con el inglés. Sin embargo, esto está bastante más allá del estado del arte actual. Así, la solución COBOL conduce a la verborrea y de hecho, es peligroso, ya que podría dar lugar a un falsa sensación de comprensión.

Por último, diremos que la legibilidad es aún más importante cuando un equipo de programadores está involucrado en un proyecto, en lugar de un programador aislado. En un equipo, es importante ser capaz de establecer convenios y reducir la cantidad de variaciones individuales, de tal forma que los programas escritos por un miembro del equipo los pueda leer otro miembro del mismo equipo. Como se mencionó en la Sección 9.1.1.1, un lenguaje que permite la expresión del mismo concepto de varias formas diferentes, anima al desarrollo de "dialectos" locales. Es inevitable que algunos programadores se acostumbren a una forma y otros a otra. Los dos grupos tendrían dificultades en la lectura de los programas del otro grupo. Las variaciones, en lugar de fomentar la creatividad, perjudicarían la legibilidad.

9.1.3 Fiabilidad

Está claro que la fiabilidad de un programa está fuertemente relacionada tanto con la facilidad de escritura (cuanto más fácil podemos escribir programas, más seguridad de corrección hay en lo que estamos escribiendo), como con la legibilidad (cuanto más fácilmente podamos leer los programas, mejor podremos razonarlos para estimar su corrección). El Capítulo 6 se dedicó por completo a la discusión de estos conceptos.

La fiabilidad se puede aumentar si el lenguaje hace una distinción rigurosa entre las comprobaciones estáticas y dinámicas que se deben realizar en los programas. Tal distinción hace a los programadores totalmente conscientes del grado de validación del programa en cada paso de su proceso. ALGOL 68 y Ada son dos ejemplos de lenguajes que sí hacen esta distinción. Por el contrario, Pascal no especifica claramente cómo y cuándo se efectúan las comprobaciones.

A causa de la falta de fiabilidad intrínseca, los lenguajes no deberían soportar características que sean imposibles o muy difíciles de comprobar. Un ejemplo se puede ver en el convenio de Ada para el paso de parámetros, el cual puede producir diferentes resultados en presencia de alias (Sección 6.2.1.2). Otro ejemplo más se puede ver en los registros variantes de Pascal (Sección 4.3.2).

Un buen objetivo de diseño, es hacer programas que se puedan comprobar lo más estáticamente posible, ya que las comprobaciones en tiempo de ejecución, por sí solas, no sólo no pueden

certificar la corrección de un programa, sino que además conlleven una mayor lentitud en la ejecución. Algunos lenguajes permiten que las comprobaciones en tiempo de ejecución se desactiven bajo petición, para hacerlos más eficientes en la ejecución, después de haber sido validados. No obstante, esta solución tiene sus propios riesgos, como agudamente señala Hoare (Hoare 1973): "Es absurdo hacer exhaustivas comprobaciones de seguridad en la depuración, cuando no se tiene confianza en los resultados, y quitarlas en la ejecución normal, cuando un resultado erróneo podría ser caro o desastroso. ¿Qué pensariamos de un navegante entusiasta que lleva salvavidas cuando se entrena en tierra, pero se lo quita tan pronto como se va al mar?".

La fiabilidad se puede aumentar con un lenguaje que permita que los programas se desarrollen y validen en módulos separados. Módulos de programas bien probados se pueden guardar en una librería del sistema y combinarlos en cualquier momento para formar nuevos programas. El esquema de compilación separada en Ada, como vimos en el Capítulo 8, es un avance en esta dirección.

La modificabilidad también contribuye a la fiabilidad, ya que en el mantenimiento debemos ser capaces de modificar el programa conservando su fiabilidad. La sintaxis del lenguaje determina en gran medida la facilidad de modificación de un programa. Una vez más, los delimitadores explícitos de Ada y ALGOL 68 tienen ventajas sobre el par begin ... end requerido por Pascal para la agrupación de sentencias. Añadir una nueva sentencia a una sentencia simple de una rama else o al cuerpo de un bucle en Pascal, requiere que se añadan un par de palabras clave delimitadoras: begin y end. Estas palabras son fáciles de olvidar, y su ausencia produciría programas incorrectos. Esto es especialmente problemático, ya que nuestro programa será, con toda probabilidad, sintácticamente válido.

Hay una fuerte relación entre la fiabilidad del programa y la definición rigurosa de la semántica del lenguaje. Además de afectar a la legibilidad del programa, como se trató en la Sección 9.1.1, y consecuentemente a la fiabilidad, una semántica formal proporciona la base para la verificación del programa (Sección 6.4.2). Un verificador de programas para el lenguaje puede convertirse en un componente importante de un sistema de programación.

Por último diremos que la fiabilidad de un lenguaje depende de la fiabilidad de su implementación. Cuanto más extenso y complejo sea el lenguaje, más difícil es producir implementaciones fiables. Ciertamente, es más fácil producir un compilador fiable de Pascal que otro igualmente fiable para PL/I o Ada.

9.2 IMPLEMENTACION DEL LENGUAJE

La implementación de un lenguaje requiere más esfuerzo que el mero diseño de un procesador para ese lenguaje. El implementador también debería disponer de "los recursos necesarios para escribir los manuales de usuario, textos introductorios y textos avanzados; debería escribir ayudas de programación auxiliares y procedimientos para bibliotecas; y finalmente, tener la voluntad política y recursos necesarios para vender y distribuir el lenguaje a los usuarios que estén dentro del ámbito para el cual se pensó el lenguaje" (Hoare 1973).

Esta sección trata brevemente el diseño de procesadores de lenguajes basados en la traducción. El diseño de un traductor es un área muy estudiada de la ciencia de los ordenadores, y gran construcción traductores fiables y eficientes. Nosotros no cubrimos estos aspectos en detalle, pero el lector interesado puede dirigirse a la literatura especializada.

Para nuestros propósitos, un procesador de un lenguaje se puede ver como una caja negra, tal como se muestra en la Figura 9.1. La entrada al procesador es un texto escrito en lenguaje fuente Lf. Su salida es la traducción del programa de entrada en un lenguaje objeto (u objetivo) Lo. El traductor es un programa escrito en un lenguaje de implementación Li y, obviamente, solamente puede ejecutarse en un ordenador anfitrión que provea un procesador para Li.

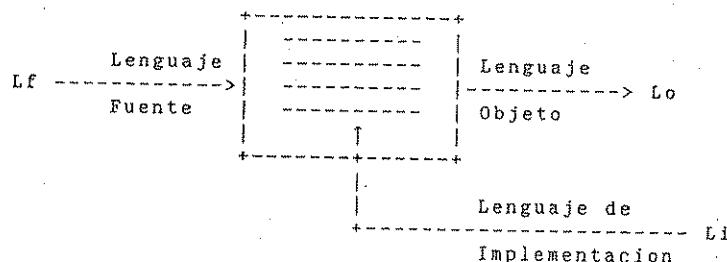


Figura 9.1 Un traductor.

Los programas objeto solamente pueden ejecutarse en ordenadores (destino u objetivo) que posean un procesador para Lo. Como se mencionó en la Sección 3.1, hay casos (traductores cruzados) en los que el ordenador destino y el anfitrión son diferentes.

La portabilidad del traductor es un objetivo de diseño deseable que implica varias ventajas. Primero, solamente se necesita desarrollar un traductor altamente fiable, y a continuación se pueden obtener muchas versiones a un coste mucho

menor. Segundo, los programas escritos en el lenguaje son automáticamente transportables a cualquier instalación que tenga el mismo traductor. Un factor que ha contribuido en gran medida a la popularidad de Pascal, es probablemente la disponibilidad de un traductor portátil, que hace que el lenguaje sea fácilmente implementable en diferentes ordenadores. C portable es un subconjunto de C que ha sido diseñado para portabilidad. El compilador original se ha transportado a muchos ordenadores destino diferentes, incluyendo microordenadores. La facilidad de producción de tal compilador para un nuevo ordenador ha causado la proliferación de estos compiladores y por lo tanto un incremento en el uso de C. SNOBOL 4 es otro ejemplo de un lenguaje que cuenta con una implementación portátil.

Las cuestiones de transportabilidad tienen una gran influencia en el diseño completo de un traductor. A continuación se trata brevemente la forma en que se ha solucionado el problema para Pascal. Un traductor Pascal (Norí y otros 1976) está disponible en los dos formatos mostrados en la Figura 9.2:

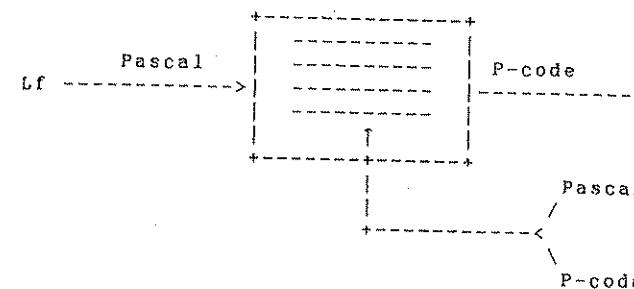


Figura 9.2 Un traductor portátil de Pascal.

uno implementado en Pascal, y el otro en "P-code". P-code es un lenguaje para una máquina virtual. Esta máquina tiene dos áreas de datos: una pila y una zona de memoria libre (heap), situadas en los extremos opuestos de una zona de datos, y creciendo una hacia la otra. Las instrucciones, almacenadas en un área separada y direccionadas por un contador de programa, constan de un código de operación y dos operandos opcionales que especifican los operandos en las áreas de datos. La mayoría de las instrucciones (p.ej., instrucciones aritméticas y lógicas) no tienen operandos explícitos, pero nombran implícitamente las posiciones de la cima de la pila. Es decir, la máquina virtual es una máquina de pila.

Para implementar un compilador Pascal en una máquina X, se debe seguir el procedimiento conocido como generación (bootstrapping):

1. Escribir un intérprete para P-code en un lenguaje disponible en X.
2. Modificar la versión Pascal del traductor mostrado en la Figura 9.2 para generar las instrucciones en lenguaje ensamblador de X. El cambio afecta únicamente a las rutinas de generación de código del compilador original. El resultado de este paso se muestra en la Figura 9.3a.
3. Utilizando el intérprete de P-code desarrollado en el paso 1, y ejecutando la versión P-code del traductor de la Figura 9.2, el traductor de la Figura 9.3a se traduce a P-code. El resultado de este paso se muestra en la Figura 9.3b.
4. El traductor de la Figura 9.3a se puede traducir ahora por medio del traductor de la Figura 9.3b utilizando el intérprete de P-code desarrollado en el paso 1. El traductor resultante (Figura 9.4) es el producto deseado.

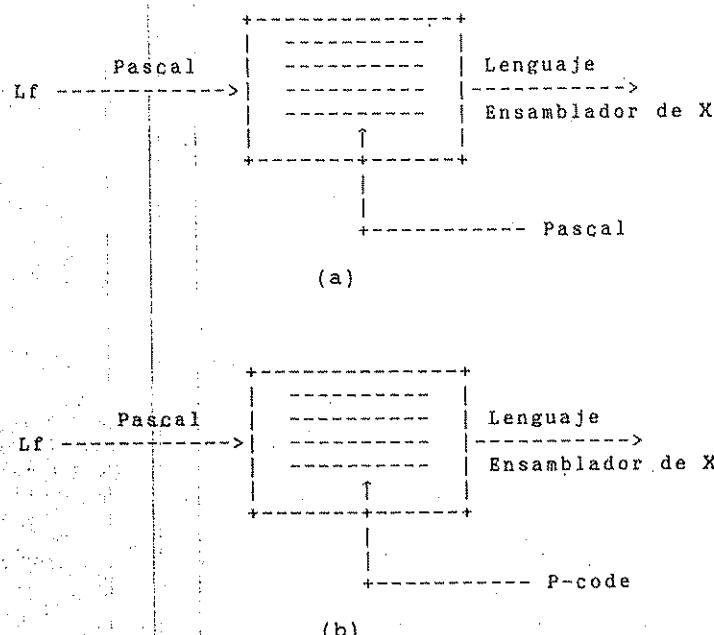


Figura 9.3 Traductores intermedios

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

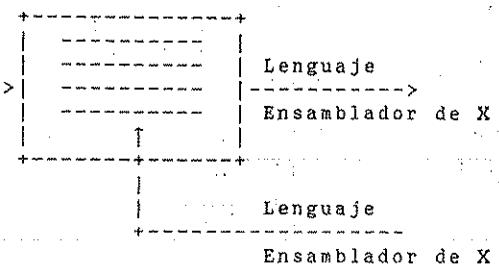


Figura 9.4 Un compilador Pascal para la máquina X.

Muchos compiladores Pascal se han instalado con éxito modificando el compilador original CDC 6600. El procedimiento de generaciones conceptualmente simple, principalmente a causa de la claridad de la estructura del compilador, lo cual permite que la modificación del programa realizada en el paso 2 sea fácil, una notable consecuencia de la legibilidad de Pascal.

SUGERENCIAS PARA AMPLIACION BIBLIOGRAFICA

Gran parte del material presentado en este capítulo está tomado de (Hoare 1973) y (Wirth 1975a). El diseño de lenguajes también se estudia en (Richman y Ledgard 1977). Los diseños de ALGOL 68 y Pascal están comparados por (Tanenbaum 1978). (Hoare y Wirth 1973) dan una definición formal de Pascal. El "Revised Report" de ALGOL 68 proporciona una definición formal del lenguaje. El desarrollo de Ada ilustra bastante bien la relación entre requisitos de los lenguajes de programación y el diseño de los mismos. El Departamento de Defensa de los Estados Unidos establece los requisitos en una serie de documentos que fueron extensamente revisados por las Fuerzas Armadas, organizaciones industriales, universidades y departamentos militares de varios países. La culminación de este proceso fue el Steelman Report (DOD 1978). Ada se eligió entre un grupo de lenguajes diseñados para reunir tales requisitos. El diseño preliminar de Ada (ACM-SIGPLAN 1979) fue revisado posteriormente, y el estándar propuesto apareció en (DOD 1980b). (Kahn y otros 1980) es un documento preliminar de la definición formal del lenguaje.

Varios libros de texto sobre diseño de compiladores tratan el tema de la implementación de los lenguajes de programación: (Gries 1971), (Bauer y Eickel 1976), (Aho y Ullman 1977) y (Barret y Couch 1979). El artículo de P.C. Poole en (Bauer y Eickel 1976) versa sobre la portabilidad de los compiladores. (Amman 1974), (Wirth 1971c) y (Nori y otros 1976) tratan la implementación de Pascal. Implementaciones portables de SNOBOL 4 y C se definen en (Griswold 1972) y (Johnson 1978) respectivamente.

Este capítulo ha estudiado el diseño de lenguajes de alto nivel independientes de la máquina. En algunas aplicaciones de software de sistemas, no se puede llevar a cabo la absoluta independencia de la máquina con una eficiencia aceptable. Una solución a este problema consiste en añadir unas pocas construcciones de alto nivel a los lenguajes ensambladores (p.ej., estructuras de control). Esta solución está bien ilustrada en PL 360 (Wirth 1968) y por varios lenguajes descritos en (Van der Poel y Maarsen 1974). Soluciones más recientes ofrecen verdaderos lenguajes de alto nivel para aplicaciones de software de sistemas. No obstante, tales lenguajes proporcionan limitadas salidas del lenguaje de alto nivel al dominio dependiente de la máquina. Bliss, C, Mesa, Euclid, Modula y PLZ siguen esta filosofía general.

Un lenguaje reciente, Edison, se describe en (Brinch Hansen 1980a). (Brinch Hansen 1980b) argumenta que incluso Pascal no es suficientemente simple, y muchas características se han eliminado o simplificado para diseñar Edison. (Brinch Hansen 1980c) da ejemplos de programas Edison.

Ejercicios

- 9.1 Algunos lenguajes de programación permiten el uso de la sobrecarga. Dé algunos ejemplos de sobrecarga en Pascal, ALGOL 68 y Ada (Ada es en particular un seguidor de la sobrecarga). Discutir los efectos de la sobrecarga en la legibilidad y en la facilidad de escritura.
- 9.2 Poner ejemplos de características no ortogonales de Ada.
- 9.3 Dé ejemplos de conversiones de tipo automáticas en los lenguajes de programación.
- 9.4 Se le dan dos traductores (ver Figura 9.5) para un nuevo lenguaje de programación (llámémosle UTOPIA 84). Suponga que la máquina M está equipada con un ensamblador y un traductor para el lenguaje X. Su objetivo es obtener un formato ejecutable del traductor (2) en la máquina M.
 - a. Muestre cómo puede obtener un formato ejecutable del traductor (2) en la máquina M.
 - b. ¿Cuándo (y por qué) es útil esta estrategia de implementación para UTOPIA 84?
 - c. Suponga que su objetivo es conseguir un traductor para UTOPIA 84 en M que produzca código optimizado. De (1) y (2), ¿cuál sería un traductor con optimización?

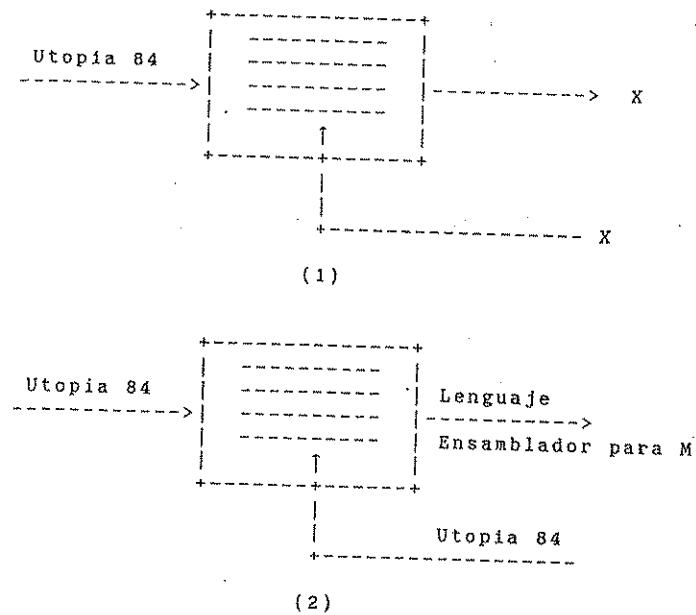


Figura 9.5 Traductores para el Ejercicio 9.4.

**GLOSARIO DE
LENGUAJES DE
PROGRAMACION
SELECCIONADOS**

ADA**Generalidades**

Ada se ha diseñado para soportar aplicaciones numéricas; programación de sistemas, y aplicaciones con exigencias de concurrencia y tiempo real.

Tipos de Datos

Ada suministra tipos predefinidos y constructores. La precisión de los datos numéricos se puede controlar por el programador. Distingue con énfasis entre las propiedades estáticas y dinámicas de los tipos. Se han eliminado muchas inseguridades del Pascal. El constructor package (verlo en "Programación a gran escala") se puede utilizar para definir tipos de datos abstractos o genéricos.

Estructuras de Control

Ada suministra estructuras de control como las de Pascal. También dispone de una sentencia exit para romper bucles, así como la sentencia goto. El lenguaje además proporciona un conjunto rico de estructuras de control a nivel de unidad; a saber: llamadas a función y procedimiento, excepciones y activaciones concurrentes.

Corrección

Ada mantiene varias características dañinas tales como goto y los efectos laterales. La utilización de alias se considera ilegal, pero pueden quedar sin detectar programas de Ada erróneos. Se describe una definición formal de Ada en (Kahn y otros 1980) y se puede utilizar como una base para la verificación de programas.

Programación a Gran Escala

En Ada, además de los subprogramas y funciones, los paquetes son las mejores unidades de estructuración de programas. Los paquetes reúnen declaraciones afines y suministran inicialización. Se pueden empaquetar varios procedimientos juntos. Las unidades se pueden anidar, como en ALGOL. Además se pueden compilar por separado de acuerdo a una regla de ordenación parcial.

Definición Oficial

(DOD 1980b), que reemplaza a (ACM-SIGPLAN 1979)

Publicaciones de Interés

(ACM-SIGPLAN 1979), (ACM-SIGPLAN 1980), (Kahn y otros 1980)

Libros de Texto

(Wegner 1980), que se refiere a (ACM-SIGPLAN 1979), y no a (DOD 1980b), (Ledgard 1981), (Pyle 1981).

ALGOL 60**Generalidades**

ALGOL 60 (Algorithmic Language 1960) es un lenguaje algebraico para cálculos científicos. Ha influido en el diseño de casi todos los lenguajes desde 1960.

Tipos de Datos

Los tipos de todos los datos se deben declarar explícitamente. Junto a los tipos predefinidos, integer, real y boolean, el lenguaje dispone de matrices dinámicas. No se soportan tipos definidos por el usuario. ALGOL 60 proporciona variables semiestáticas y semidinámicas.

Estructuras de Control

ALGOL 60 suministra if-then-else y bucles controlados por contador. También se suministra el goto y la bifurcación múltiple. Las llamadas a procedimientos suponen la única estructura de control a nivel de unidad. Se permiten llamadas recursivas.

Corrección

Este no fue un objetivo de diseño. El lenguaje suministra varias características nocivas. Esta disponible el goto. Las llamadas a funciones y procedimientos pueden generar efectos laterales, puesto que llaman por nombre, lo cual es estandar, y acceden a variables globales. La sintaxis del lenguaje se ha definido formalmente, pero su semántica se ha descrito sólo en prosa informal inglesa.

Programación a Gran Escala

Los procedimientos y los bloques son los únicos mecanismos de abstracción disponibles en ALGOL 60. Los programas están estructurados en forma de árbol, y las unidades internas heredan automáticamente las declaraciones externas.

Declaración Oficial

(Naur 1963)

Publicaciones de Interés

(Knuth 1967), (Perlis 1978), (Naur 1978)

Libros de Texto

(Dijkstra 1962); ver también (Sammet 1969).

Notas de Implementación

(Randell y Russell 1964), (Gries 1971), (Wichmann 1973).

ALGOL 68**Generalidades**

ALGOL 68 se diseñó para plasmar algoritmos, ejecutarlos eficientemente sobre una variedad de ordenadores, y para ayudar a enseñárselos a los estudiantes.

Tipos de Datos

ALGOL 68 suministra tipos predefinidos y constructores para crear nuevos tipos. El lenguaje no soporta tipos abstractos de datos. La compatibilidad de tipos se define por equivalencia estructural. Suministra unas extensas conversiones de tipo automáticas (coerciones). La compatibilidad de tipos está rigurosamente definida.

Estructuras de Control

ALGOL 68 proporciona if-then-else, case, bucle generalizado y goto. Las llamadas a procedimiento gobiernan el flujo de control entre las unidades. Se pueden pasar procedimientos como parámetros. Soporta programación concurrente y semáforos para la sincronización.

Corrección

Se suministra la sentencia goto y permite efectos laterales de una forma limitada y controlada. En la definición oficial del lenguaje, la semántica se definió en parte formalmente, y parcialmente en inglés informal. La semántica de ALGOL 68 también se ha definido axiomáticamente en (Schwartz 1978b), en un intento de ayudar a la verificación de programas.

Programación a Gran Escala

Las únicas capacidades de abstracción son los bloques, los procedimientos, y los operadores definidos por el usuario. Los programas forman una estructura de árbol y los módulos internos heredan automáticamente las declaraciones externas. La definición oficial del lenguaje no especifica una facilidad estándar de compilación separada.

Publicaciones de Interés

(Branquart y otros 1971), (Lindsay 1972), (Tanenbaum 1976),
(Valentine 1974), (Tanenbaum 1978).

Libros de Texto

(Lindsay y van der Meulen 1977), (Pagan 1976).

Notas de Implementación

(Peck 1970), (Hill 1976), (Branquart y otros 1976).

APL**Generalidades**

APL (A Programming Language) se diseñó como una notación para expresar concisamente algoritmos matemáticos. APL está soportado por un sistema de programación interactivo que ha contribuido a su popularidad entre los programadores científicos que utilizan el computador como un potente calculador de mesa, y también entre usuarios de otro tipo.

Tipos de Datos

APL es un lenguaje de tipos dinámicos. Los tipos predefinidos son numérico, carácter y lógico. Suministra un conjunto rico de operaciones sobre matrices, que eliminan la necesidad de manipular elemento a elemento para la mayoría de las operaciones. Las matrices son el único método de estructuración y obtención de agregados.

Estructuras de Control

APL suministra unas reglas de composición muy potentes para las operaciones, y los programas en APL frecuentemente toman la forma de transatlánticos "one-liners". El lenguaje suministra únicamente unas estructuras de control rudimentarias, incluyendo el salto.

Corrección

Este no fué un objetivo del APL. La legibilidad de los programas se ve deteriorada a causa de las ligaduras dinámicas. La ejecución interactiva proporciona un mecanismo de prueba, que es especialmente válido para programas que no son demasiado grandes.

Programación a Gran Escala

Los programas en APL se pueden descomponer en subprogramas. Las referencias no locales se basan en ligaduras de ámbito dinámico, que pueden dañar la legibilidad y modificabilidad de los programas grandes. APL está soportado por un sistema de programación completo.

Definición Oficial

(Falkoff e Iverson 1978), (Iverson 1979), (Iverson 1980).

Libros de Texto

(Polivka y Pakin 1975)

Notas de Implementación

(Breed y Lathwell 1968)

BLISS**Generalidades**

Bliss es un lenguaje de alto nivel, orientado a expresiones para escribir software de sistemas. Utilizado actualmente con este propósito en ordenadores DEC.

Tipos de Datos

No tiene tipos de datos. Las estructuras de datos se definen no por su distribución de memoria, sino más bien en términos del algoritmo utilizado para acceder a un elemento de la estructura.

Estructuras de Control

Bliss es un lenguaje estructurado en bloques, que no dispone de la sentencia goto y que está orientado a expresiones. Cada construcción ejecutable es una expresión. Las expresiones se pueden concatenar con puntos y comas para formar sentencias. Bliss además suministra sentencias condicionales, bucles, sentencia leave (para ruptura de bucles), y subprogramas (denominados routines). También dispone de construcciones para el manejo de excepciones.

Corrección

Bliss ofrece ventajas sobre los lenguajes ensambladores, que podrían utilizarse para las mismas aplicaciones. Sin embargo, la corrección formal de programas no fué un objetivo de diseño.

Programación a Gran Escala

Bliss tiene una estructura convencional como ALGOL. Tiene bloques y rutinas, las cuales pueden ser externas.

Definición Oficial

(Wulf y otros 1972)

Publicaciones de Interés

(Wulf y otros 1971)

Notas de Implementación

(Wulf y otros 1975)

C**Generalidades**

C se desarrolló para escribir programas de sistemas para el PDP-11. Se ha generalizado e implementado en muchos ordenadores, tanto grandes como pequeños. Dispone de un conjunto completo de herramientas de soporte en el sistema operativo UNIX. También se ha definido un subconjunto transportable. C se basa en el BCPL (Richards 1969), un lenguaje para la escritura de compiladores.

Tipos de Datos

Los tipos de datos primitivos se pueden formar con matrices, estructuras y uniones (similares a la unión de ALGOL 68). Se puede especificar la precisión de los números enteros y de los reales. Las conversiones de tipo se aplican libre y automáticamente. Los punteros se califican.

Estructuras de Control

Se suministran el repeat, while y bucles, con la posibilidad de salida del bucle, break, o salto del resto de la iteración del bucle actual, y continue. También están disponibles una forma limitada de la sentencia case y del goto general. Los procedimientos y funciones son las únicas estructuras de control de bloques.

Corrección

No se dedicó especial atención en este sentido. Están presentes el goto, punteros, efecto lateral y conversión automática de tipos. La legibilidad puede llegar a ser un problema, debido a la existencia de pocas formas distintas de reflejar el mismo concepto y la posibilidad de poder producir un código fuente extremadamente sucinto. La sintaxis de las declaraciones de tipos es bastante criptica. La definición es

informal y deja bastantes cuestiones sin resolver.

Programación a Gran Escala

Los procedimientos y funciones son los únicos mecanismos de programación estructurada. No se pueden anidar, aunque pueden incluir bloques anidados. Los procedimientos y las funciones se pueden compilar por separado, pero el compilador no hace chequeo de tipos entre módulos.

Declaración Oficial

(Kernighan y Ritchie 1978).

Libros de Texto

(Kernighan y Ritchie 1978), (Zahn 1979).

Notas de Implementación

(Johnson 1978).

CLU**Generalidades**

CLU se diseñó para soportar un método de programación basado en el reconocimiento de las abstracciones.

Tipos de Datos

CLU dispone de tipos predefinidos y constructores. El constructor cluster le permite al programador definir tipos abstractos de datos. A los objetos se accede uniformemente a través de punteros. La ejecución de una sentencia de asignación cambia la referencia a un objeto, en lugar de cambiar el valor de la posición de memoria que referenciaba antes de ejecutar la sentencia.

Estructuras de Control

CLU suministra las estructuras de control convencionales ya establecidas. No hay goto, sólo existe una sentencia break para salir de los bucles. Además, el programador puede definir nuevas estructuras de control de bucles mediante iteradores. Este lenguaje dispone de facilidades para el manejo de excepciones.

Corrección

Se han evitado muchas características dañinas, tales como el goto y las variables globales. Sin embargo, la asignación por compartición estimula el uso excesivo de efectos laterales. La semántica se definió en prosa informal inglesa. No se hace mención explícita de la verificación, aunque está llamada a ser

soportada por el lenguaje.

Programación a Gran Escala

Los iteradores, procedimientos y agrupaciones (clusters) suministran unas bases cómodas para la descomposición en sistemas modulares. Los procedimientos e iteradores se pueden declarar dentro de las agrupaciones, pero no se permite ninguna otra anidación. Los iteradores, procedimientos y agrupaciones se pueden compilar por separado y en cualquier orden. Cuando se compila un módulo, sólo necesitan estar ya compilados los módulos a los que se haga referencia.

Definición Oficial

(Liskov y otros 1978).

Publicaciones de Interés

(Liskov y Zilles 1974), (Liskov 1974), (Liskov y Zilles 1975), (Liskov y otros 1977), (Liskov y Snyder 1979).

Notas de Implementación

(Atkinson y otros 1978).

COBOL

Generalidades

COBOL (Common Business Oriented Language) es un lenguaje para aplicaciones comerciales. Los programas COBOL normalmente realizan unos cálculos muy simples sobre un conjunto grande de datos. Quizás sea el lenguaje de uso más extendido hoy día, pero no jugó un papel significativo en el diseño de los lenguajes posteriores.

Tipos de Datos

Los datos se describen en la DATA DIVISION. Casi todos los datos se agrupan como componentes de registros que se almacenan en un fichero. COBOL tiene instrucciones para manejo de registros y ficheros. Los tipos de los componentes individuales de los registros y las variables elementales es necesario declararlas. Los tipos básicos son las cadenas de caracteres y los números de precisión especificada por el programador.

Estructuras de Control

COBOL suministra una forma restringida del IF-THEN-ELSE, la sentencia GOTO, y una sentencia PERFORM que sirve tanto como sentencia de bucle como de llamada a una unidad. El PERFORM suministra una transferencia de control a la unidad, y retorno de la misma, sin cambiar el entorno. Las versiones más antiguas de

COBOL no disponen de subprogramas. Lo cual también sucede en muchas implementaciones de subconjuntos de COBOL.

Corrección

Este no fué un objetivo en el diseño de COBOL. Para facilitar el uso del lenguaje, COBOL introduce una programación con el estilo del lenguaje natural. Los programas se hacen fácilmente verbales y los errores son difíciles de encontrar. La sintaxis se definió formalmente, no así su semántica.

Programación a Gran Escala

Un programa COBOL consiste de una IDENTIFICATION DIVISION, que identifica al programador y al programa; una ENVIRONMENT DIVISION, que especifica la configuración hardware y la relación entre los ficheros lógicos y físicos; una DATA DIVISION, que especifica la estructura de los datos; y una PROCEDURE DIVISION, que especifica los algoritmos que operan sobre los datos. No se puede conseguir una modularidad real dentro de la PROCEDURE DIVISION.

Definición Oficial

(ANSI 1968), (ANSI 1974).

Publicaciones Importantes

(Sammet 1978).

Libros de Texto

Lo esencial de COBOL se puede encontrar en (Rosen 1969) y (Sammet 1969).

EUCLID

Generalidades

Euclid es un lenguaje diseñado para escribir programas de sistemas verificables.

Tipos de Datos

La estructura de tipos de Euclid se basa en Pascal, pero se omiten muchas características inseguras de Pascal. El campo etiqueta de los registros con variante no se puede asignar por sí mismo. Los punteros se ligán a colecciones. Aunque la compatibilidad de tipos está rigurosamente definida, el programador puede obligar explícitamente a que se ignore el chequeo de tipos en tiempo de compilación, si así lo necesita. Los tipos abstractos de datos se pueden implementar por módulos; ver más adelante "Programación a Gran Escala".

Estructuras de Control

Euclid suministra estructuras de control como Pascal. No hay goto, pero se suministra una sentencia exit para salir de los bucles. El constructor module se puede utilizar para definir nuevas estructuras de control de bucles, similares a los iteradores de CLU. Las estructuras de control de bloques son las llamadas a procedimientos y funciones.

Corrección

Euclid no dispone de goto. Las funciones no pueden producir efectos laterales. Los programas correctos sintácticamente permiten la generación potencial de alias, y el compilador genera asertos adecuados que se deben verificar para certificar la corrección de un programa, con ayuda de un verificador que se incluye en el sistema. El verificador se encarga de comprobar que se cumplen los asertos de legalidad, tanto los establecidos por el usuario como los generados por el compilador.

Programación a Gran Escala

Al lado de los procedimientos y funciones, Euclid suministra un constructor module que se puede utilizar para definir tanto abstracciones de datos como de control. Los módulos son similares a los packages de Ada. No se pueden anidar, pero hay control sobre las entidades globales importadas. Los módulos externos se pueden compilar por separado.

Definición Oficial

(Lampson y otros 1977).

Publicaciones de Interés

(Popek y otros 1977), (Elliott y Barnard 1978), (London y otros 1978), (Wortman 1979).

Notas de Implementación

(Holt y otros 1978b).

FORTRAN

Generalidades

FORTRAN (Formula Translator) es un lenguaje para aplicaciones científicas y numéricas. Fue diseñado con el objetivo principal de una ejecución eficiente. A continuación se esboza el FORTRAN estándar descrito en (ANSI 1966). Un nuevo estándar está redactado en (ANSI 1978).

Tipos de Datos

FORTRAN dispone de booleanos, enteros, reales (simple y doble precisión), y números complejos. La matriz de tamaño fijo es el único constructor de agregados. Los datos se pueden declarar explícita o implícitamente. No soporta tipos definidos por el usuario.

Estructuras de Control

FORTRAN proporciona la sentencia lógica IF (con una sola sentencia en la rama THEN), bifurcaciones con tres alternativas, bucles por contador, y GOTOS. Las llamadas a subprogramas son las únicas estructuras de control a nivel de unidad que se permiten.

Corrección

Este no era un objetivo del lenguaje. FORTRAN tiene varias características dañinas. El GOTO se debe utilizar a menudo, debido a la carencia de estructuras de control apropiadas, por lo que es fácil que se utilice indebidamente. El alias es posible (por medio de la sentencia EQUIVALENCE). Los subprogramas pueden generar efectos laterales manipulando parámetros reales (pasados por referencia) o variables globales (especificadas por COMMON). El lenguaje está definido en prosa informal inglesa.

Programación a Gran Escala

FORTRAN soporta estructuras de programa sin anidamientos. Un programa es una colección de subprogramas externos que pueden compartir datos comunes (COMMON). Los subprogramas se pueden compilar independientemente, y ensamblarse para construir un sistema. Normalmente no proporciona comprobación de tipos entre los módulos.

Definición Oficial

(ANSI 1966), (ANSI 1971), (ANSI 1978).

Publicaciones de Interés

(Backus 1957), (Backus 1978b), (Brainerd 1978).

Libros de Texto

Lo fundamental de FORTRAN se puede encontrar en (Rosen 1967) y (Sammet 1969).

Notas de Implementación

(Gries 1971).

Gypsy**Generalidades**

Gypsy es un lenguaje diseñado para soportar la especificación, codificación y verificación del software de sistemas, con énfasis particular en el software de comunicaciones.

Tipos de Datos

Gypsy cuenta con tipos predefinidos y constructores. No proporciona explícitamente variables de tipo puntero. En su lugar, cuenta con algunos tipos de datos completamente dinámicos, tales como secuencias y aplicaciones. Una unidad Gypsy puede contener una definición de tipo, y una lista de acceso específica los accesos legales (ver más adelante "Programación a Gran Escala"). De esta forma, es posible implementar tipos abstractos de datos.

Estructuras de Control

Gypsy dispone de estructuras de control de tipo Pascal. No existe el goto, pero sí hay una sentencia leave para salir de los bucles. Las estructuras de control a nivel de unidad son las llamadas a procedimientos y a funciones, excepciones, e invocaciones concurrentes.

Corrección

Este lenguaje no soporta una sentencia goto. Las funciones no pueden modificar los parámetros reales. No hay variables globales ni punteros. Se han excluido la mayoría de las características dañinas. Un verificador de programas forma parte integral del sistema Gypsy.

Programación a Gran Escala

Las unidades del programa son las rutinas (procedimiento, función o proceso) y la definición de constantes y de tipos. Las unidades no están anidadas, teniendo cada una de ellas una lista de accesos que establece los accesos correctos a la unidad. No existen las variables globales. Las unidades se verifican y compilan de forma separada.

Definición Oficial del Lenguaje

(Ambler y otros 1976).

Publicaciones de Interés

(Ambler y otros 1977), (Good 1977).

Notas de Implementación

(Good y otros 1978).

LISP**Generalidades**

LISP (List Processing) es un lenguaje funcional. Se usa en la mayoría de las aplicaciones de inteligencia artificial. Existen varios dialectos que amplian LISP con características no funcionales.

Tipos de Datos

LISP tiene dos tipos de datos: átomos y listas. Proporciona funciones para operar con listas. Los programas están uniformemente representados por estructuras de listas, de tal forma que la evaluación de un programa se puede describir por un intérprete (función EVAL) que transforma en valores una estructura de lista que representa un programa.

Estructuras de Control

El LISP puro no tiene ni sentencias de asignación ni sentencias GOTO. Es un lenguaje puramente funcional, fuertemente basado en la recursión. Dialectos de LISP proporcionan ambas sentencias, la de asignación y la GOTO.

Corrección

LISP fué el primer lenguaje que se diseñó sobre una base puramente matemática: (McCarthy 1963a), (McCarthy 1963b).

Programación a Gran Escala

Esto no es uno de los principios de LISP. Muchos sistemas LISP soportan el desarrollo de programas con variedad de herramientas.

Definición Oficial

McCarthy y otros 1965). Dos dialectos LISP están definidos por (Moon 1974) y (Teitelman 1975).

Publicaciones de Interés

(McCarthy 1960), (McCarthy 1963a), (McCarthy 1963b), (McCarthy 1978), (Sandewall 1978).

Libros de Texto

(Siklossy 1976), (Allen 1978).

Artículos de Implementación)

(McCarthy y otros 1965), (Henderson 1980).

Mesa**Generalidades**

Mesa es un componente de un sistema de programación pensado para el desarrollo y mantenimiento de un amplio espectro de programas de sistemas y aplicaciones.

Tipos de Datos

Mesa proporciona los tipos predefinidos convencionales, enumeraciones, subrangos, y constructores (matriz, registro con variante y puntero). Mesa es muy rígido en el manejo de tipos, pero el programador puede desinhibir explícitamente la comprobación de tipos. El lenguaje provee varias conversiones automáticas de tipos. Los módulos Mesa (ver más adelante "Programación a Gran Escala") soportan la definición de tipos abstractos de datos.

Estructuras de Control

Las estructuras de control a nivel de sentencia incluyen la sentencia if, la sentencia select (muy parecida al case de Pascal), sentencias de bucle, salida de bucle, y goto. Las estructuras de control a nivel de unidad incluyen las llamadas a subprogramas, activación de corrutinas, condiciones de excepción, y activaciones concurrentes (la comunicación entre procesos está soportada por monitores).

Corrección

El lenguaje cuenta con características seguras, pero también permite el uso de algunas perjudiciales, tales como goto y la eliminación de la comprobación de tipos. Este lenguaje está definido en prosa informal inglesa, y no habla explícitamente de lo referente a la verificación.

Programación a Gran Escala

Mesa proporciona la construcción module para encapsular abstracciones. Cada módulo tiene una definición, la cual especifica el interfaz del módulo, y un programa, que contiene datos y código ejecutable. Las definiciones no existen en ejecución, pero se permite compilar programas separadamente con una completa comprobación de tipos. Los programas se pueden cargar e interconectar, para formar sistemas completos, por medio de comandos escritos en el lenguaje de configuración de Mesa.

Definición Oficial

(Mitchell y otros 1979).

Publicaciones de Interés

(Geschke y Mitchell 1975), (Lampson y otros 1974), (Geschke y otros 1977).

Modula**Generalidades**

Modula es un lenguaje de programación dedicado a sistemas de ordenadores, incluyendo control de procesos en máquinas más pequeñas. Proporciona una visibilidad limitada del hardware subyacente. Este lenguaje está muy basado en Pascal.

Tipos de Datos

Modula adopta la mayoría de los conceptos de Pascal en lo concerniente a los tipos de datos. La excepción más notable es la ausencia de punteros. Los tipos abstractos de datos se pueden definir por medio de la construcción module (ver más adelante "Programación a Gran Escala").

Estructuras de Control

Este lenguaje adopta las estructuras de control de Pascal con mínimas variaciones. También dispone de bucles con salidas por el medio de ellos. Proporciona procedimientos y funciones. Se pueden inicializar explícitamente unidades ejecutables concurrentemente (procesos). La sincronización se consigue mediante señales. Una señal se puede enviar, y un proceso puede esperar por ella. (Las señales son similares a las colas de Pascal Concurrente). Los módulos de interfaz (correspondientes a los monitores) son secciones de código ejecutados con exclusión mutua.

Corrección

El lenguaje está pensado para cubrir aplicaciones para las que tradicionalmente se viene empleando lenguaje ensamblador. Modula mejora considerablemente la legibilidad de tales programas, y soporta una gran variedad de comprobaciones estáticas. En la actualidad no hay disponible ninguna definición formal del lenguaje. (Wirth 1977) habla del uso de Modula en lo concerniente a pruebas de corrección de programas de tiempo real.

Programación a Gran Escala

Modula mantiene la estructura de bloques de tipo Pascal. Además proporciona la construcción module. Un módulo es una colección de declaraciones y una parte de inicialización, muy parecido al package de Ada. Un módulo puede "importar" entidades

explicativamente, y "exportarlas" al resto del programa.

Definición Oficial

(Wirth 1976b); revisada en (Wirth 1978).

Publicaciones de Interés

(Wirth 1976c), (Wirth 1977), (Wirth 1979).

Notas de Implementación

(Wirth 1976d).

Pascal

Generalidades

Pascal se diseñó en un principio para la enseñanza de programación disciplinada. El lenguaje se ha encontrado con un gran éxito, y ahora existen implementaciones para la mayoría de las máquinas, incluso para las basadas en microprocesadores. Pascal ha influido en casi todos los lenguajes de programación más recientes.

Tipos de Datos

Dispone de tipos predefinidos, tipos por enumeración, subrangos, y constructores (registro, registro con variantes, matriz, fichero, puntero, y conjunto) para generar nuevos tipos. La compatibilidad de tipos no está rigurosamente definida y tampoco soporta tipos abstractos de datos.

Estructuras de Control

Este lenguaje proporciona if-then-else, while-do, repeat-until, bucles for, y la sentencia goto. Las únicas estructuras de control a nivel de unidad son las llamadas a procedimientos y funciones.

Corrección

Pascal permite los efectos laterales y la utilización de la sentencia goto. La semántica del lenguaje se ha definido axiomáticamente (Hoare y Wirth 1973), sin embargo, los axiomas para el lenguaje no se han desarrollado en su totalidad.

Programación a Gran Escala

La única posibilidad de abstracción se soporta por medio de los procedimientos y las funciones. Los programas tienen una estructura de árbol y los módulos internos heredan automáticamente las declaraciones más externas. La declaración oficial del lenguaje no especifica la facilidad estándar de la

compilación separada.

Definición Oficial

(Jensen y Wirth 1975).

Publicaciones de Interés

(Wirth 1971a), (Hoare y Wirth 1973), (Wirth 1975b), (Habermann 1973), (Lecarme y Desjardins 1975), (Welsh y otros 1977), (Tennent 1978), (Tanenbaum 1978).

Libros de Texto

(Wirth 1973), (Wirth 1976a), (Findlay y Watt 1978), (Alagić y Arbib 1978).

Notas de Implementación

(Wirth 1971c), (Ammann 1974), (Nori y otros 1976).

PASCAL CONCURRENTE

Generalidades

Pascal concurrente es un lenguaje para escribir programas concurrentes estructurados, en particular, sistemas operativos. Es una extensión de Pascal.

Tipos de Datos

El lenguaje suministra los tipos de Pascal y el tipo class para definir tipos abstractos de datos. Los tipos process y monitor son las herramientas básicas para describir la concurrencia. A los tipos class, process y monitor se les denomina en conjunto "tipos del sistema".

Estructuras de Control

El lenguaje suministra las estructuras de control de sentencias de Pascal. Además, un tipo process puede repetir indefinidamente la ejecución de un conjunto de sentencias mediante una sentencia cycle. Las estructuras de control de bloques son las llamadas a procedimientos y funciones, así como la activación concurrente de procesos (por la sentencia init).

Corrección

Soporta la escritura de programas concurrentes legibles bien estructurados, así como un extenso chequeo estático. En particular, garantiza, a nivel estático, que no se den interbloqueos. También posibilita la verificación de programas (Hoare 1974), (Howard 1976a), (Howard 1976b).

Programación a Gran Escala

Un programa consta de definiciones anidadas de tipos del sistema. El tipo del sistema más externo es un proceso anónimo llamado "proceso inicial". El proceso inicial se activa después de la carga del programa y posteriormente inicializa el resto de los componentes. El lenguaje soporta la descomposición de sistemas modulares mediante una estructura de árbol como en ALGOL.

Definición Oficial

(Brinch Hansen 1975).

Publicaciones de Interés

(Hoare 1974), (Howard 1976a), (Howard 1976b).

Libros de Texto

(Brinch Hansen 1977).

Notas de Implementación

(Hartmann 1977).

PL/I**Generalidades**

PL/I (Programming Language I) representa un intento de incorporar en un único lenguaje multipropósito las características más notables de los anteriores lenguajes de programación (FORTRAN, ALGOL 60, y COBOL).

Tipos de Datos

PL/I proporciona tipos predefinidos para los que se pueden especificar una gran variedad de atributos (p.ej. base y precisión). Los constructores de agregados de datos incluyen estructuras (registro), matrices, y punteros. La asignación de memoria a las variables se puede efectuar estéticamente (tipo FORTRAN), automáticamente (tipo ALGOL 60), o bien explícitamente.

Estructuras de Control

Soporta IF_THEN_ELSE, WHILE_DO, bucles por contador, y GOTO. Las estructuras de control a nivel de unidad incluyen la llamada a subprogramas, manejo de excepciones y soporte de multitarea (para unidades concurrentes).

Corrección

No es precisamente uno de sus objetivos fundamentales. Este lenguaje contiene varias características perjudiciales. El GOTO está permitido; los punteros no tienen asociado el tipo de los objetos a los que apuntan, por lo que se puede dar el caso de referenciar el valor de un objeto cuya dirección no es la que nosotros creemos (p.ej. si se nos olvidó inicializar o actualizar el puntero en el que está basado el objeto). Las llamadas a subprogramas pueden generar efectos laterales y el alias está permitido. También se han definido subconjuntos para el área de la educación y prácticas de programación: PL/C en (Conway y Gries 1979); PLCS en (Conway 1978); y SP/K en (Conway y otros 1977). El lenguaje se definió formalmente por investigadores del Laboratorio IBM de Viena. A este método se le conoce como Vienna Definition Language: (Lucas y Walk 1969), Wegner (1972). PL/C también está definido formalmente, y además existe un verificador de programas que está descrito en (Constable y O'Donnell 1978).

Programación a Gran Escala

Un programa se puede estructurar tipo FORTRAN, es decir, como un conjunto de subprogramas externos. Cada subprograma puede estar estructurado, como en ALGOL 60, como un conjunto de unidades anidadas.

Definición Oficial

(ANSI 1976).

Publicaciones de Interés

(Lucas y Walk 1969), (Radin 1978).

Libros de Texto

(Conway y Gries 1979). Lo fundamental se puede encontrar en (Sammet 1969).

Notas de Implementación

(Abrahams 1979).

PLZ**Generalidades**

PLZ fue diseñado por Zilog Corporation para facilitar la construcción de programas para sistemas de microcomputadores. El diseño del lenguaje estuvo fuertemente influenciado por Pascal. Las partes de un programa que son críticas en el tiempo o necesitan acceder explícitamente a dispositivos de la máquina, se pueden escribir en un "PLZ bajo-nivel" (llamado PLZ/ASM).

Tipos de Datos

PLZ tiene tipos predefinidos (byte, palabra, entero-corto, y entero), y constructores (registro, matriz y puntero). La compatibilidad de tipos está muy claramente definida. Se dispone de conversiones explícitas de tipos para permitir una violación controlada de la comprobación de tipos.

Estructuras de Control

Las estructuras de control a nivel de sentencia incluyen *if_then_else*, *case*, y *loop*. Los bucles se pueden finalizar por medio de un *exit* explícito. También es posible salir de una iteración activa (sentencia *repeat*).

Corrección

Mejora la fiabilidad de programas relacionados con aplicaciones que tradicionalmente se estaba forzado a codificar en ensamblador. No se dispone de características dañinas del bajo nivel (p.ej. *goto*). La semántica está definida informalmente. La verificación de programas no se trata explícitamente.

Programación a Gran Escala

Un programa PLZ es un conjunto de módulos, siendo un módulo la unidad básica de compilación. Un módulo está formado por declaraciones de datos y de procedimientos. Los procedimientos y datos que se quieran tener disponibles para su uso en otros módulos se deben especificar explícitamente. Los módulos soportan diseño de programas por ocultamiento de la información ("information hiding").

Definición Oficial

(Snook y otros 1978).

Publicaciones de Interés

(Crespi-Reghizzi y otros) los cuales han estudiado otros lenguajes de alto nivel para microprocesadores.

Libros de Texto

(Conway y otros 1980).

Notas de Implementación

(Snook y otros 1978).

SIMULA 67

Generalidades

SIMULA 67 es un lenguaje de propósito general cuya principal área de aplicación está en la simulación.

Tipos de Datos

Además de los tipos predefinidos y matrices, SIMULA 67 dispone de clases para la definición de tipos abstractos de datos. Pueden existir simultáneamente varias clases en tiempo de ejecución. Se pueden asignar por referencia (:-), y además la notación puntual (.) proporciona el acceso a componentes individuales. Los detalles de representación no quedan escondidos en la "clase".

Estructuras de Control

Además de las estructuras de control a nivel de sentencia de tipo ALGOL y llamadas a procedimientos, SIMULA 67 tiene corrutinas para simular ejecuciones concurrentes.

Corrección

Tiene la sentencia *goto* y efectos laterales que deterioran la legibilidad del programa. La semántica está descrita en un inglés informal. La verificación de programas no fué un objetivo en el diseño de este lenguaje.

Programación a Gran Escala

Soporta las abstracciones por medio de procedimientos. Se puede implementar la abstracción de datos por medio de clases. Los prefijos de clases permiten la definición de estructuras jerárquicas de programas. En general, este lenguaje mantiene la estructura de programas en árbol como en ALGOL. No se define oficialmente un esquema de compilación por separado.

Definición Oficial

(Dahl y otros 1970).

Publicaciones de Interés

(Ichbiah y Morse 1972), (Dahl y Hoare 1972), (Nygaard y Dahl 1978).

Libros de Texto

(BirtWistle y otros 1973).

Artículos de Implementación

(Dahl y Myhrhaug 1969).

SNOBOL4**Generalidades**

SNOBOL4 (String Oriented Symbolic Language) es un lenguaje apropiado para el manejo de "strings". Su principal aplicación reside en áreas en las que datos que son cadenas de caracteres que se deben procesar de modos complejos, por ejemplo, en el proceso de textos de lenguaje natural.

Tipos de Datos

Es un lenguaje con tipos dinámicos. Posee potentes operaciones sobre los "strings" para el reconocimiento de modelos. También soporta la definición de nuevos tipos de datos. Por medio de su TABLE de tipos de datos, proporciona una forma de acceso por asociación. Los "strings" generados en ejecución se pueden tratar como código y ser ejecutados como tal.

Estructuras de Control

Las estructuras de control a nivel de sentencia son bastante simples. Las más complejas residen en el control de reconocimiento de modelos. A los subprogramas se les puede llamar recursivamente. Este lenguaje soporta también el manejo de condiciones de excepción. El manejo de excepciones se utiliza básicamente para la generación de trazas en la fase de depuración de los programas.

Corrección

Esto no es un objetivo del lenguaje. A causa de los enlaces dinámicos, se deteriora la legibilidad si los programas son grandes. La facilidad de las trazas ayuda en las pruebas del programa. Gran parte de la semántica de SNOBOL4 está definida formalmente en (Tennent 1973).

Programación a Gran Escala

Los programas se pueden descomponer en subprogramas. No obstante, la definición del subprograma es una operación en tiempo de ejecución. Esto es otro ejemplo de los enlaces dinámicos establecidos por el lenguaje. Tales enlaces dificultan la realización de programas extensos.

Definición Oficial

(Griswold y otros 1971).

Publicaciones de Interés

(Griswold 1978), (Tennent 1973).

Libros de Texto

(Griswold y otros 1971), (Griswold y Griswold 1973).

Notas de Implementación

(Griswold 1972) describe una implementación transportable.

DONACIONAL CHAVEZ 286486
 \$.....
 Fecha 16.05.05
 Inv. E..... Inv. B..... 1457

BIBLIOGRAFIA

- (Abrahams 1979)
P. Abrahams, "The CIMS PL/I Compiler." Proceedings SIGPLAN Symp. on Compiler Construction-SIGPLAN Notices 14.8 (Agosto 1979).
- (ACM-CS 1974)
ACM Computing Surveys, número especial: Programming 6.4 (Diciembre 1974).
- (ACM-SIGPLAN 1976)
Proceedings of Conference on Data: Abstraction, Definition and Structure. SIGPLAN Notices 8.2 (1976).
- (ACM-SIGPLAN 1977)
Proceedings ACM Conference on Language Design for Reliable Software. SIGPLAN Notices 12.3 (Marzo 1977).
- (ACM-SIGPLAN 1978)
ACM-SIGPLAN History of Programming Languages Conference. SIGPLAN Notices 13.8 (Agosto 1978).
- (ACM-SIGPLAN 1979)
J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Bruckner, y B.A. Wichmann. "Preliminary Ada Reference Manual" y "Rationale for the Design of the Ada Programming Language." SIGPLAN Notices 14.6, Partes A y B (Junio 1979).
- (ACM-SIGPLAN 1980)
Proceedings ACM-SIGPLAN Symposium on the Ada Programming Language. SIGPLAN Notices 15.11 (Noviembre 1980).
- (Aho y Ullman 1977)
A.V. Aho y J.D. Ullman. Principles of Compiler Design. Reading, Mass.: Addison-Wesley, 1977.
- (Alagic y Arbib 1978)
S. Alagic y M.A. Arbib. The Design of Well-Structured and Correct Programs. New York: Springer Verlag, 1978.
- (Alford 1977)
M.W. Alford. "A Requirements Engineering Methodology for Real-Time Processing Requirements." IEEE Transactions on Software Engineering SE-3.1 (Enero 1977): 60-68.
- (Allen 1978)
J. Allen. Anatomy of LISP. New-York: McGraw-Hill, 1978.
- (Ambler y otros 1976)
A. Ambler, D.I. Good, y W.F. Burger. "Report on the Language Gypsy." Univ. de Texas en Austin, ICSA-CMP-1 (Agosto 1976).

(Ambler y otros 1977)

A.L. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch, y R.E. Wells. "Gypsy: A Language for Specification and Implementation of Verifiable Programs." Proceedings ACM Conference on Language Design for Reliable Software-SIGPLAN Notices 12 3 (Marzo 1977): 1-10.

(Ammann 1974)

U. Ammann. "The Method of Structured Programming Applied to the Development of a Compiler." En Proceedings ACM Int. Comp. Symp., editores A. Gunther y otros Amsterdam: North-Holland, 1974.

(ANSI 1966)

American National Standard FORTRAN (ANS X3.9-1966) New York: American National Standards Institute, 1966.

(ANSI 1968)

USA Standard COBOL (ANS X3.23-1968). New York: American National Standards Institute, 1968.

(ANSI 1971)

"Clarification of FORTRAN Standard-Second Report." Comm. ACM 14 10. (Octubre 1971): 628-642.

(ANSI 1974)

American National Standard Programming Language COBOL (ANS X3.23-1974). New York: American National Standard Institute, 1974.

(ANSI 1976)

American National Standard Programming Language PL/I (ANS X3.53-1976). New York: American National Standard Institute, 1976.

(ANSI 1978)

American National Standard Programming Language FORTRAN (ANS X3.9-1978). New York: American National Standard Institute, 1978.

(Aron 1974)

J.D. Aron. The Program Development Process, Part I: The Individual Programmer. Reading, Mass.: Addison-Wesley, 1974.

(Ashcroft y Wagde 1977)

E.A. Ashcroft y W.W. Wagde. "LUCID: A Nonprocedural Language with Iteration." Comm. ACM 20 7 (Julio 1977): 519-526.

(Asirelli y otros 1979)

P. Asirelli, P. Degano, G. Levi, A. Martelli, U. Montanari, G. Pacini, F. Sirovich, y F. Turini. "A Flexible Environment for Program Development Based on a Symbolic Interpreter." Proceedings 4th Int. Conference on Software Engineering. IEEE Cat. No. 79CH1479-5C (Munich, Septiembre 1979): 251-263.

(Atkinson y otros 1978)

R.R. Atkinson, B.H. Liskov, y R.W. Scheifler. "Aspects of Implementing CLU." Proceedings ACM National Conference 1 (Diciembre 1978): 123-129.

(Backus 1957)

Ver (Rosen 1967).

(Backus 1973)

J. Backus. "Programming Language Semantics and Closed Applicative Languages." Conf. Record ACM Symp. on Principles of Programming Languages (Boston, Octubre 1973) 71-86.

(Backus 1978a)

J. Backus. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs." Comm. ACM 21 8 (Agosto 1978): 613-641.

(Backus 1978b)

Ver (ACM-SIGPLAN 1978).

(Baker 1972)

F.T. Baker. "Chief Programmer Team Management of Production Programming." IBM System Journal Enero 1972: 56-73.

(Barret y Couch 1979)

W.A. Barrett y J.D. Couch. Compiler Construction: Theory and Practice. Chicago: SRA, 1979.

(Bauer y Eickel 1976)

F.L. Bauer y J. Eickel, editores. Compiler Construction: An Advanced Course, segunda edición. New York: Springer Verlag, 1976.

(Berkling 1976)

K.J. Berkling. "Reduction Languages for Reduction Machines." ISF-76-8 GMB (Bonn, Septiembre 1976).

(Berry 1979)

Ver (Wegner 1979).

(Birtwistle y otros 1973)

G.M. Birtwistle, O-J. Dahl, B. Myhrhaug, y K. Nygaard. SIMULA Begin. New York: Petrocelli/Charter, 1973.

(Birtwistle y otros 1976)

G. Birtwistle, L. Enderin, M. Ohlin, y J. Palme. "DEC System-10 SIMULA Language Handbook - Part 1." En Report N. C8398. Stockholm: Swedish National Defense Research Institute, Marzo 1976.

(Björner y Jones 1978)

D. Björner y C.B. Jones, editores. The Vienna Development Method: The Meta Language. Lecture Notes in Computer Science 61. New York: Springer Verlag, 1978.

- (Bobrow y Raphael 1974)
D.G. Bobrow y B. Raphael. "New Programming Languages for Artificial Intelligence." *ACM Computing Surveys* 6. (1972): 155-174.
- (Boehm 1976)
B. Boehm. "Seven Basic Principles of Software Engineering." En Infotech State of the Art Report on Software Engineering Techniques. Maidenhead, U.K.: Infotech International Ltd., 1976.
- (Böhm y Jacopini 1966)
C. Böhm y G. Jacopini. "Flow-diagrams, Turing Machines, and Languages with Only Two Formation Rules." *Comm. ACM* 9 5 (Mayo 1966): 366-371.
- (Brainerd 1978)
W. Brainerd, editor. "FORTRAN 77." *Comm. ACM* 21 10 (Octubre 1978): 806-820.
- (Branquart y otros 1971)
P. Branquart, J. Lewi, M. Sintzoff, y P. Wodon. "The Composition of Semantics in ALGOL 68." *Comm. ACM* 14 11 (Noviembre 1971): 697-708.
- (Branquart y otros 1976)
P. Branquart, J.-P. Cardinael, J. Lewi, J.-P. Delescaillie, y M. Vanbegin. An Optimized Translation Process and its Application to ALGOL 68. Lecture Notes in Computer Science 38. New York: Springer Verlag, 1976.
- (Breed y Lathwell 1968)
L.M. Breed y R.H. Lathwell. "The Implementation of APL/360." En Symposium on Interactive Systems for Experimental and Applied Mathematics, editores: Klerer y Reinfelds. New York: Academic Press, 1968.
- (Brinch Hansen 1973)
P. Brinch Hansen. Operating Systems Principles. Englewood Cliffs, N.J.: Prentice-Hall, 1973.
- (Brinch Hansen 1975)
P. Brinch Hansen. "The Programming Language Concurrent Pascal." *IEEE Transactions on Software Engineering* SE-1 2 (Junio 1975): 199-207.
- (Brinch Hansen 1977)
P. Brinch Hansen. The Architecture of Concurrent Programs. Englewood Cliffs, N.J.: Prentice-Hall, 1977.
- (Brinch Hansen 1979)
P. Brinch Hansen. "Distributed Processes: A Concurrent Programming Concept." *Comm. ACM* 21 11 (Noviembre 1979): 934-941.
- (Brinch Hansen 1980a)
P. Brinch Hansen. Edison - A Multiprocessor Language. Los Angeles, Cal.: Dept. of Computer Science, Univ. of Southern California, Septiembre 1980.
- (Brinch Hansen 1980b)
P. Brinch Hansen. The Design of Edison. Los Angeles, Cal.: Dept. of Computer Science, Univ. of Southern California, Septiembre 1980.
- (Brinch Hansen 1980c)
P. Brinch Hansen. Edison Programs. Los Angeles, Cal.: Dept. of Computer Science, Univ. of Southern California, Septiembre 1980.
- (Brooks 1975)
F.P. Brooks, Jr. The Mythical Man-Month - Essays on Software Engineering. Reading, Mass.: Addison-Wesley, 1975.
- (Burge 1975)
W.H. Burge. Recursive Programming Techniques. Reading, Mass.: Addison-Wesley, 1975.
- (Buxton 1980)
J.N. Buxton. "An Informal Bibliography on Programming Support Environments." *ACM SIGPLAN Notices* 15 12 (Diciembre 1980): 17-30.
- (Cashin y otros 1981)
P.M. Cashin, M.L. Joliat, R.F. Kamel y D.M. Lasker. "Experience with a Modular Typed Language: PROTEL." Proceedings 5th International Conf. Software Engineering (Marzo 1981): 136-143.
- (Celentano y otros 1980)
A. Celentano, P. Della Vigna, C. Ghezzi y D. Mandrioli. "Separate Compilation and Partial Specification in Pascal." *IEEE Transactions on Software Engineering* SE-6 4 (Julio 1980): 313-319.
- (Cheatham 1977)
T.E. Cheatham, Jr. "Some New Directions in Program Development Tools." AICA Proceedings (Italian Computer Society) 3 (Pisa, Octubre 1977): 3-29.
- (Cheatham y otros 1979)
T.E. Cheatham, Jr.; J.A. Townley y G.H. Holloway. "A System for Program Refinement." Proceedings 4th Int. Conf. on Software Engineering. IEEE cat. no. 79CH1479-5C (Munich, Septiembre 1979): 53-62.
- (Clarke y otros 1980)
L.A. Clarke, J.C. Wileden y L. Wolf. "Nesting in Ada is for the Birds." Proceedings Symposium on the Ada Programming Language - SIGPLAN Notices 15 11 (Noviembre 1980).

- (Constable y O'Donnell)
R.L. Constable y M.J. O'Donnell. *A Programming Logic with an Introduction to the PL/CV Verifier*. Cambridge, Mass.: Winthrop, 1978.
- (Conway 1963)
M.E. Conway. "Design of Separable Transition-Diagram Compiler." *Comm. ACM* 6 7 (Julio 1963): 396-408.
- (Conway 1978)
R. Conway. *A Primer on Disciplined Programming*. Cambridge, Mass.: Winthrop, 1978.
- (Conway y Gries 1979)
R. Conway y D. Gries. *An Introduction to Programming - A Structured Approach Using PL/I and PL/C*. Tercera Edición. Cambridge, Mass.: Winthrop, 1979.
- (Conway y otros 1977)
R. Conway, D. Gries y D.B. Wortman. *Introduction to Structured Programming Using PL/I and SP/k*. Cambridge, Mass.: Winthrop, 1977.
- (Conway y otros 1979)
R. Conway, D. Gries, M. Fay y C. Bass. *Introduction to Microprocessor Programming Using PLZ*. Cambridge, Mass.: Winthrop, 1979.
- (Crespi-Reghizzi y otros 1980)
S. Crespi-Reghizzi, P. Corti y A. Daprà. "A Survey of Microprocessor Languages." *Computer* 13 1 (Enero 1980) 48-66.
- (Dahl y Hoare 1972)
Ver (Dahl y otros 1972).
- (Dahl y Myhrhaug 1969)
O.-J. Dahl y B. Myhrhaug. "SIMULA 67 Implementation Guide." Publication N. S-9. Oslo: Norwegian Computing Center, Junio 1969.
- (Dahl y otros 1970)
O.-J. Dahl, B. Myhrhaug y K. Nygaard. "SIMULA 67 Common Base Language." Publication N. S-22. Oslo: Norwegian Computing Center, Octubre 1970.
- (Dahl y otros 1972)
O.-J. Dahl, E.W. Dijkstra y C.A.R. Hoare. *Structured Programming*. New York: Academic Press, 1972.
- (Darlington y Burstall 1976)
J. Darlington y R.M. Burstall. "A System Which Automatically Improves Programs." *Acta Informatica* 6 1 (1976): 41-60.
- (Demers y Donahue 1980a)
A. Demers y J. Donahue. "Data Types, Parameters and Type Checking." *Conference Record of the 7th Annual ACM Symp. on Principles of Programming Languages*. (Enero 1980): 12-23.
- (Demers y Donahue 1980b)
A. Demers y J. Donahue. "Type Completeness as a Language Principle." *Conference Record of the 7th Annual ACM Symp. on Principles of Programming Languages*. (Enero 1980): 234-244.
- (DeMillo y otros 1979)
R.A. DeMillo, R.J. Lipton y A.J. Perles. "Social Processes and Proof of Theorems and Programs." *Comm. ACM* 22 5 (Mayo 1979): 271-280.
- (DeRemer y Kron 1976)
F. DeRemer y H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small." *IEEE Transactions on Software Engineering SE-2* (Junio 1976): 80-86.
- (Deutsch y Bobrow 1976)
L.P. Deutsch y D.G. Bobrow. "An Efficient Incremental Automatic Garbage Collector." *Comm. ACM* 19 9 (Septiembre 1976): 522-526.
- (Dijkstra 1962)
E.W. Dijkstra. *A Primer of ALGOL 60 Programming*. New York: Academic Press, 1962.
- (Dijkstra 1968a)
E.W. Dijkstra. "Goto Statement Considered Harmful." *Comm. ACM* 11 3 (Marzo 1968): 147-149.
- (Dijkstra 1968b)
E.W. Dijkstra. "Cooperating Sequential Processes." *In Programming Languages*, editor F. Genuys. New York: Academic Press, 1968.
- (Dijkstra 1968c)
E.W. Dijkstra. "The Structure of The Multiprogramming System." *Comm. ACM* 11 5 (Mayo 1968): 341-346.
- (Dijkstra 1972)
Ver (Dahl y otros 1972).
- (Dijkstra 1976)
E.W. Dijkstra. *A Discipline of Programming*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.
- (Dijkstra y otros 1978)
E.W. Dijkstra, B. Lamport, A.J. Martin, C.S. Scholten y S.F.M. Steffens. "On-the-Fly Garbage Collection: An Exercise in Cooperation." *Comm. ACM* 21 11 (Noviembre 1978): 966-975.

- (DOD 1977) United States Department of Defense. Requirements for High Order Computer Programming Languages, Revised "Ironman." SIGPLAN Notices 12 12 (Diciembre 1977): 39-54.
- (DOD 1978) United States Department of Defense. Requirements for High Order Computer Programming Languages, "Steelman". Junio 1978.
- (DOD 1980a) United States Department of Defense. "Stoneman": Requirements for Ada Programming Support Environment. Febrero 1980.
- (DOD 1980b) United States Department of Defense. Reference Manual for the Ada Programming Language. Proposed standard document. Julio 1980.
- (Elliot y Barnard 1978) W.D. Elliot y D.T. Barnard, editores. "Notes on Euclid." SIGPLAN Notices 13 3 (Marzo 1978): 34-89.
- (Elson 1973) M. Elson. Concepts of Programming Languages. Chicago: SRA, 1973.
- (Falkoff e Iverson 1978) Ver (ACM-SIGPLAN 1978).
- (Findlay y Watt 1978) W. Findlay y D.A. Watt. Pascal: An Introduction to Methodical Programming. Potomac, Md.: Computer Sciences Press, 1978.
- (Fischer y LeBlanc 1980) G.N. Fischer y R.J. LeBlanc. "The Implementation of Run-Time Diagnostics in Pascal." IEEE Transactions on Software Engineering SE-6 4 (Julio 1980): 313-319.
- (Floyd 1967) R.W. Floyd. "Assigning Meanings to Programs." Proc Symp. Appl. Math. En Mathematical Aspects of Computer Science, editor J.T. Schwartz. Providence, R.I.: American Mathematical Society, 1967.
- (Fosdick y Osterweil 1976) L.D. Fosdick y L.J. Osterweil. "Data Flow Analysis in Software Reliability." Computer Surveys 8 3 (Octubre 1976): 305-330.

- (Foxall y otros 1979) D.G. Foxall, M.L. Filiat, R.F. Kamel y J.J. Miceli. "PROTEL: A High Level Language for Telephony." Proc. 3rd International Computer Software and Applications Conf. (Noviembre 1979): 193-197.
- (Francez 1977) N. Francez. "Another Advantage of Keyword Notation for Parameter Communication with Subprograms." Comm. ACM 20 8 (Agosto 1977): 604-605.
- (Gannon 1977) J.D. Gannon. "An Experimental Evaluation of Data Types Conventions." Comm. ACM 20 8 (Agosto 1977): 584-595.
- (Gannon y Horning 1975) J.D. Gannon y J.J. Horning. "Language Design for Programming Reliability." IEEE Transactions on Software Engineering SE-1 2 (1975): 179-191.
- (Geschke y Mitchell 1975) C. Geschke y J. Mitchell. "On the Problem of Uniform References to Data Structures." IEEE Transactions on Software Engineering SE-1 2 (Junio 1975): 207-219.
- (Geschke y otros 1977) C.M. Geschke; J.H. Morris, Jr; E.H. Satterwhite. "Early Experience with Mesa." Comm. ACM 20 8 (Agosto 1977): 540-553.
- (Goguen y otros 1978) Ver (Yeh 1978).
- (Good 1977) D.I. Good, editor. Constructing Verifiably Reliable and Secure Communications Processing Systems. Univ. of Texas at Austin, ICSCA-CMP-6 (Enero 1977).
- (Good y otros 1978) D.I. Good, R.M. Cohen y L.W. Hunter. "A Report on the Development of Gypsy." Proceedings of ACM National Conference 1 (Diciembre 1978): 116-122.
- (Goodenough 1975) J.B. Goodenough. "Exception Handling: Issues and Proposed Notation." Comm. ACM 16 12 (Diciembre 1975): 683-696.
- (Goodenough) Ver (Wegner 1979).
- (Gordon 1979) R. Gordon. The Notational Description of Programming Languages. New York: Springer Verlag, 1979.

- (Gries 1971)
D. Gries. *Compiler construction for Digital Computers*. New York: J. Wiley, 1971.
- (Gries y Gehani 1977)
D. Gries y N. Gehani. "Some Ideas on Data Types in High Level Languages." *Comm. ACM* 20 6 (Junio 1977): 414-420.
- (Griswold 1972)
R.E. Griswold. *The Macroimplementation of SNOBOL4*. San Francisco: W.H. Freeman, 1972.
- (Griswold 1978)
Ver (ACM-SIGPLAN 1978).
- (Griswold y Griswold 1973)
R.E. Griswold y M.T. Griswold. *A SNOBOL4 Primer*. Englewood Cliffs, N.J.: Prentice-Hall, 1973.
- (Griswold y otros 1971)
R.E. Griswold, J.F. Poage e I.P. Polonsky. *The SNOBOL4 Programming Language*, (segunda edición). Englewood Cliffs, N.J.: Prentice-Hall, 1971.
- (Guarino 1978)
L.R. Guarino. "The Evolution of Abstraction in Programming Languages." *Carnegie-Mellon Univ. Dept. of Computer Science Report CMU-CS-78-120* (Mayo 1978).
- (Guttag 1977)
J.V. Guttag. "Abstract Data Types and the Development of Data Structures." *Comm. ACM* 20 6 (Junio 1977): 396-404.
- (Guttag y otros 1978)
Ver (Yeh 1978).
- (Habermann 1973)
A.N. Habermann. "Critical Comments of the Programming Language Pascal." *Acta Informatica* 3 (1973): 47-57.
- (Habermann 1975)
A.N. Habermann. *Introduction to Operating System Design*. Chicago: SRA, 1976.
- (Habermann 1979)
A.N. Habermann. *Carnegie Mellon University - Computer Science Research Review 1978-79*. 1979.
- (Hartmann 1977)
A.C. Hartmann. *A Concurrent Pascal Compiler for Minicomputers. Lecture Notes in Computer Science* 50, New York: Springer Verlag, 1977.
- (Hecht 1977)
M.S. Hecht. *Flow Analysis of Computer Programs*. New York: Elsevier North-Holland, 1977.
- (Henderson 1980)
P. Henderson. *Functional Programming: Application and Implementation*. Englewood Cliffs, N.J.: Prentice-Hall, 1980.
- (Hill 1976)
Ver (Bauer y Eickel 1976).
- (Hoare 1969)
C.A.R. Hoare. "An Axiomatic Basis of Computer Programming." *Comm. ACM* 12 10 (Octubre 1969): 576-580.
- (Hoare 1972a)
Ver (Dahl y otros 1972).
- (Hoare 1972b)
C.A.R. Hoare. "Proof of Correctness of Data Representation." *Acta Informatica* 1 (1972): 271-281.
- (Hoare 1973)
C.A.R. Hoare. "Hints on Programming Language Design." Notas de la ACM SIGACT/SIGPLAN Conference on Principles of Programming Languages. Boston: Octubre 1973. Ver también el Stanford Univ. Computer Science Dept. Tech. Rep. STAN-CS-74-403.
- (Hoare 1974)
C.A.R. Hoare. "Monitors: An Operating System Structuring Concept." *Comm. ACM* 17 10 (Octubre 1974): 549-557.
- (Hoare 1975a)
C.A.R. Hoare. "Data Reliability." *Proceedings Intl. Conf. on Reliable Software - SIGPLAN Notices* 10 6 (Junio 1975): 528-533.
- (Hoare 1975b)
C.A.R. Hoare. "Recursive Data Structures." *Int. Journal of Comp. and Inf. Sciences* 4 2 (1975): 105-132.
- (Hoare 1978)
C.A.R. Hoare. "Communicating Sequential Processes." *Comm. ACM* 21 8 (Agosto 1978): 666-677.
- (Hoare y Wirth 1973)
C.A.R. Hoare y N. Wirth. "An Axiomatic Definition of the Programming Language Pascal." *Acta Informatica* 2 (1973): 335-355.
- (Holt y otros 1978a)
R.C. Holt, G.S. Graham, E.D. Lazowska y M.A. Scott. *Structured Concurrent Programming with Operating System Applications*. Reading, Mass.: Addison-Wesley, 1978.

- (Holt y otros 1978b)
R.C. Holt, D.B. Wortman, J.R. Cordy, D.R. Crowe. "The Euclid Language: A Progress Report." *Proceedings of the ACM National Conference 1* (Diciembre 1978): 111-115.
- (Horowitz 1975)
E. Horowitz, editor. *Practical Strategies for Developing Software Systems*. Reading, Mass.: Addison-Wesley, 1975.
- (Howard 1976a)
J.H. Howard. "Proving Monitors." *Comm. ACM 19 5* (Mayo 1976): 273-279.
- (Howard 1976b)
J.H. Howard. "Signaling in Monitors." *Proceedings of the 2nd. Intl. Conference on Software Engineering. IEEE cat. no. 76CH1125-4C* (San Francisco, Octubre 1976): 47-52.
- (Ichbiah y Morse 1972)
J.D. Ichbiah y S.P. Morse. "General Concepts of the SIMULA 67 Programming Language." *Annual Review of Automatic Programming I* (1972): 65-95.
- (Ichbiah y otros 1979)
Ver (ACM-SIGPLAN 1979).
- (Ingalls 1978)
D.H. Ingalls. "The Smalltalk-76 Programming System Design and Implementation." *Conference Record of the 5th. Annual ACM Symp. on Principles of Programming Languages* (Enero 1978): 9-16.
- (Iverson 1962)
K.E. Iverson. *A Programming Language*. New York: J. Wiley, 1962.
- (Iverson 1979)
K.E. Iverson. "Operators." *ACM Transactions on Programming Languages and Systems 1 2* (Octubre 1979): 161-176.
- (Iverson 1980)
K.E. Iverson. "Notations as a Tool of Thought." *Comm. ACM 23 8* (Agosto 1980): 444-465.
- (Jackson 1975)
M.A. Jackson. *Principles of Program Design*. New York: Academic Press, 1975.
- (Jensen y Wirth 1975)
K. Jensen y N. Wirth. *Pascal User Manual and Report*. New York: Springer Verlag, 1975.
- (Johnson 1978)
S.C. Johnson. "A Portable Compiler: Theory and Practice." *Conf. Record of 5th Annual ACM Symp. on Principles of Programming Languages* (Enero 1978): 97-104.
- (Johnston 1971)
J. Johnston. "The Contour Model of Block-Structured Processes." *Proceedings Symp. Data Structures in Programming Languages-SIGPLAN Notices 6 2* (Febrero 1971): 55-82.
- (Kahn y otros 1980)
G. Kahn, V. Donzeau-Gouge, B. Lang. "Formal Definition of the Ada Programming Language." *Honeywell-Bull INRIA Report*. (Noviembre 1980).
- (Kennedy y Schwartz 1975)
K. Kennedy y J. Schwartz. "An Introduction to the Set Theoretical Language SETL." *Journal Computer and Math. with Applications 1* (1975): 97-119.
- (Kernighan y Masey 1979)
B.W. Kernighan y J.R. Masey. "The UNIX Programming Environment." *Software-Practice and Experience 9 1* (1979): 1-16.
- (Kernighan y Plauger 1976)
B.W. Kernighan y P.J. Plauger. *Software Tools*. Reading, Mass.: Addison-Esley, 1976.
- (Kernighan y Ritchie 1978)
B.W. Kernighan y D.M. Ritchie. *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall, 1978.
- (Kieburz 1976)
Ver (ACM-SIGPLAN 1976).
- (Kieburz y otros 1978)
R.B. Kieburz, W. Barabash, y S.R. Hill. "A Type-Checking Program Linkage System for Pascal." *Proceedings 3rd Int. Conf. on Software Eng.* Atlanta Ga.: Mayo 10-12, 1978.
- (King 1976)
J.C. King. "Symbolic Execution and Program Testing." *Comm. ACM 19 7* (Julio 1976): 385-394.
- (Knuth 1967)
D.E. Knuth. "The Remaining Trouble Spots in ALGOL 60." *Comm. ACM 10 10* (Octubre 1967): 611-617.
- (Knuth 1973)
D.E. Knuth. *The Art of Computer Programming*. Vol. 1: *Fundamental Algorithms*, segunda edición. Reading, Mass.: Addison-Wesley, 1973.

(Knuth 1974)

D.E. Knuth. "Structured Programming with GOTO Statements." *ACM Computing Surveys* 6 4 (Diciembre 1974): 261-301.

(Lampson y otros 1974)

B. Lampson, J. Mitchell, y E. Satterhwaite. "On the Transfer of Control between Contexts." *Lecture Notes in Computer Science* 19: 181-203. New York: Springer-Verlag, 1974.

(Lampson y otros 1977)

B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, y G.J. Popek. "Report on the Programming Language Euclid." *SIGPLAN Notices* 12 2 (Febrero 1977). (Revised Report, XEROX PARC Tech. Rep. CSL78-2.)

(Landin 1966)

P.J. Landin. "The Next 700 Programming Languages." *Comm. ACM* 9 3 (Marzo 1966): 157-164.

(Lasker 1979)

D.M. Lasker. "Module Structure in an Evolving Family of Real Time Systems." *Proc. 4th Int'l. Conf. Software Engineering* IEEE cat. no. 79 CH1479-5C (Munich 1979) (Septiembre 1979): 22-28.

(Lauer y Satterhwaite 1979)

H.C. Lauer y E.H. Satterhwaite. "The Impact of Mesa on System Design." *Proc. 4th Int'l. Conf. Software Engineering*. IEEE cat. no. 79 CH1479-5C (Munich 1979): 174-182.

(LeBlanc y Fischer 1979)

R.J. LeBlanc y C.N. Fischer. "On Implementing Separate Compilation Block-Structured Languages." *Proceedings SIGPLAN Symp. on Compiler Construction-SIGPLAN Notices* 14 8 (Agosto 1979): 133-143.

(Lecarme y Desjardins)

O. Lecarme y P. Desjardins. "More Comments on the Programming Language Pascal." *Acta Informatica* 4 (1975): 231-243.

(Ledgard 1981)

H. Ledgard. Ada: An introduction (Part 1) Ada Reference Manual -- Julio 1980 (Part 2), New York, N.Y.: Springer Verlag, 1981.

(Levin 1977)

R. Levin. Program Structures for Excepcional Condition Handling. Ph.D. dissertation. Carnegie-Mellon Univ. Dept. of Computer Science, Junio 1977.

(Lindsey 1972)

C.H. Lindsey, "ALGOL 68 with Fewer Tears." *Computer Journal*. 15 (1972): 176-188.

(Lindsey y van der Meulen 1977)

C.H. Lindsey y S. G. van der Meulen. *Informal Introduction to ALGOL 68*, rev. ed. Amsterdam: North-Holland, 1977.

(Liskov 1974)

B.H. Liskov. "A Note on CLU." *Computation Structures Group Memo* 112. Cambridge, Mass.: MIT Project MAC, Noviembre 1974.

(Liskov y Snyder)

B.H. Liskov y A. Snyder. "Exception Handling in CLU." *IEEE Trans. on Software Engineering* 5 6 (Noviembre 1979): 547-558.

(Liskov y Zilles 1974)

B.H. Liskov y S.N. Zilles. "Programming with Abstract Data Types." *SIGPLAN Symp. on Very High Level Languages - SIGPLAN Notices* 9 4 (Abril 1974): 50-59.

(Liskov y Zilles 1975)

B.H. Liskov y S.N. Zilles. "Specification Techniques for Data Abstractions." *IEEE Trans. on Software Engineering* SE-1 1 (1975): 7-19.

(Liskov y otros 1977)

B.H. Liskov, A. Snyder, R. Atkinson y C. Schaffert. "Abstraction Mechanisms in CLU." *Comm. ACM* 20 8 (Agosto 1977): 564-576.

(Liskov y otros)

B. Liskov, E. Moss, C. Schaffert, R. Scheiffer y A. Snyder. "CLU Reference Manual." *Computation Structures Group Memo* 161. Cambridge Mass.: Massachusetts Institute of Technology Laboratory for Computer Science, Julio 1978.

(London 1979)

Ver (Wegner 1979).

(London y otros 1978)

R.L. London, J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell y G.J. Popek. "Proof Rules for the Programming Language Euclid." *Acta Informatica* 10 (1978): 1-26.

(Lucas y Walk 1969)

P. Lucas y K. Walk. "On the Formal Description of PL/I." *Annual Review of Automatic Programming* 6 3 (1969): 105-182.

(MacLaren 1977)

D.M. MacLaren. "Exception Handling in PL/I." *Proceedings Conference on Language Design for Reliable Software - SIGPLAN Notices* 12 3 (Marzo 1977): 101-104.

(Magó 1980)

G.A. Magó. "A Network of Microprocessors to Execute Reduction Languages." *Int. J. Computer Sci.* 1980.

- (Manna 1973)
Z. Manna. *The Mathematical Theory of Computation*. New York: McGraw-Hill, 1973.
- (Marlin 1980)
C.D. Marlin. *Couroutines*. Lecture Notes in Computer Science 95. New York: Springer Verlag, 1980.
- (McCarthy 1960)
J. McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine." *Comm. ACM* 3 4 (Abril 1960): 184-195.
- (McCarthy 1963a)
J. McCarthy. "A Basis for a Mathematical Theory of Computation." En *Computer Programming and Formal Systems*, editores P. Braffort y D. Hirschberg: 33-37. Amsterdam: North-Holland, 1963.
- (McCarthy 1963b)
J. McCarthy. "Towards a Mathematical Science of Computation." *Proceedings of IFIP Congress (Munich)*: 21-28. Amsterdam: North-Holland, 1963.
- (McCarthy y otros 1965)
J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart y M.I. Levin. *LISP 1.5 Programmer's Manual*. Segunda edición. Cambridge, Mass.: The MIT Press, 1965.
- (McCarthy 1978)
Ver (ACM-SIGPLAN 1978).
- (Metzger 1973)
P.W. Metzger. *Managing a Programming Project*. Englewood Cliffs, N.J.: Prentice-Hall, 1973.
- (Mitchell y otros 1979)
J.G. Mitchell, W. Maybury y R. Sweet. *Mesa Language Manual (Version 5.0)*. Xerox Research Center, Palo Alto, Cal.: CSL-79-3 (Abril 1979).
- (Moon 1974)
D.A. Moon. *MACLISP Reference Manual*. Project MAC Technical Report. Cambridge, Mass.: Massachusetts Institute of Technology, 1974.
- (Myers 1975)
G.J. Myers. *Reliable Software through Composite Design*. New York: Petrocelli/Charter, 1975.
- (Myers 1976)
G.J. Myers. *Software Reliability: Principles and Practices*. New York: J. Wiley, 1976.
- (Myers 1978)
G.J. Myers. *Composite/Structured Design*. New York: Van Nostrand Reinhold, 1978.
- (Myers 1979)
G.J. Myers. *The Art of Software Testing*. New York: J. Wiley, 1979.
- (Naur 1963)
P. Naur, editor. "Revised Report on the Algorithms Language ALGOL 60." *Comm. ACM* 6 1 (Enero 1963): 1-17. Ver también (Rosen 1967).
- (Naur 1978)
Ver (ACM-SIGPLAN 1978).
- (Nori y otros 1976)
K.V. Nori, U. Ammann, K. Jensen, H.H. Nageli y Ch. Jacobi. *The Pascal "P" Compiler: Implementation Notes*. Edición revisada. Berichte Nr. 10. Zurich: Institut für Informatik, Eidgenössische Technische Hochschule, 1976.
- (Nygaard y Dahl 1978)
Ver (ACM-SIGPLAN 1978).
- (Organick y otros 1978)
E.I. Organick, A.I. Forsythe y R.P. Plummer. *Programming Language Structures*. New York: Academic Press, 1978.
- (Pagan 1976)
F.P. Pagan. *Practical Guide to ALGOL 68*. New York: J. Wiley, 1976.
- (Parnas 1972a)
D.L. Parnas. "A Technique for Module Specification with Examples." *Comm. ACM* 15 5 (Mayo 1972): 330-336.
- (Parnas 1972b)
D.L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules." *Comm. ACM* 15 12 (Diciembre 1972): 1053-1058.
- (Parnas 1975)
D.L. Parnas. "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering SE-2 1* (Marzo 1975): 1-9.
- (Parnas 1977)
Ver (Yeh 1977).
- (Parnas y Würges 1976)
D.L. Parnas y H. Würges. "Response to Undesired Events in Software Systems." *Proceedings 2nd. Int'l. Conference on Software Engineering*. IEEE cat. no. 76CH1125-4C: 437-443. (San Francisco, Cal., 13-15 Octubre, 1976).

(Peck 1970)

J.E.L. Peck, editor. ALGOL 68 Implementation. Amsterdam: North-Holland, 1970.

(Perlis 1978)

Ver (ACM-SIGPLAN 1978).

(Polivka y Pakin 1975)

R.P. Polivka y S. Pakin. APL: The Language and Its Usage. Englewood Cliffs, N.J.: Prentice-Hall, 1975.

(Popek y otros 1977)

G.J. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell y R.L. London. "Notes on the Design of Euclid." Proceedings ACM SIGPLAN Notices 12 3 (Marzo 1977): 11-18.

(Pozefsky 1977)

M. Pozefsky. "Programming in Reduction Languages." Ph.D. dissertation. Univ. of North Carolina Computer Science Dept., 1977.

(Prat 1975)

T.W. Pratt. Programming Languages: Design and Implementation. Englewood Cliffs, N.J.: Prentice-Hall, 1975.

(Pyle 1981)

I.C. Pyle. The Ada Programming Language. Englewood Cliffs, N.J.: Prentice-Hall, 1981.

(Radin 1978)

Ver (ACM-SIGPLAN 1978).

(Randell 1975)

B. Randell. "System Structure for Software Fault Tolerant." IEEE Transactions on Software Engineering SE-1 2 (Junio 1975): 220-232.

(Randell y Russell 1964)

B. Randell y L. Russell. ALGOL 60 Implementation. New York: Academic Press, 1964.

(Reynolds 1970)

J.C. Reynolds. "GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept." Comm. ACM 13 5 (Mayo 1970): 305-319.

(Reynolds 1979)

J.C. Reynolds. "Syntactic Control of Interference." Proc. Fifth Annual ACM Symp. on Principles of Programming Languages. Tucson, Ariz.: 23-25 Enero, 1979.

(Richard y Ledgard 1977)

F. Richard y H.F. Ledgard. "A Reminder for Language Designers." SIGPLAN Notices 12 12 (Diciembre 1977): 73-82.



(Richards 1969)

M. Richards. "BCPL: A tool for Compiler and System Writing." Spring Joint Comp. Conference (1969): 557-566.

(Ritchie y Thompson 1974)

D. Ritchie y K. Thompson. "The Unix Time-Sharing System." Comm. ACM 17 7 (Julio 1974): 365-375.

(Rosen 1967)

S. Rosen. Programming Systems and Languages. Ney York: McGraw-Hill, 1967.

(Sammet 1969)

J.E. Sammet. Programming Languages: History and Fundamentals. Englewood Cliffs, Prentice-Hall, 1969.

(Sammet 1978)

Ver (ACM-SIGPLAN 1978).

(Sandewall 1978)

E. Sandewall. "Programming in the Interactive Environment: The Lisp Experience." ACM Computing Surveys 10 1 (Marzo 1978): 35-71.

(Schorr y Waite 1967)

H. Schorr y W. Waite. "An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures." Comm. ACM 10 8 (Agosto 1967): 501-506.

(Schwartz 1974)

J.T. Schwartz. On Programming: An Interim Report on the SETL Project. New York: Courant Inst. Math. Sci. of New York Univ., 1974.

(Schwartz 1978a)

R.L. Schwartz. "Paralell Compilation: A Design and Its Application to SIMULA 67." Journal of Computer Languages 3 (1978): 75-94.

(Schwartz 1978b)

R.L. Schwartz. "An Axiomatic Semantic Definition of ALGOL 68." Ph.D. dissertation. Univ. of Calif. en Los Angeles Computer Science Dept. Rep. no. UCLA-ENG-7838, Julio 1978.

(Shaw y otros 1977)

M. Shaw, W.A. Wulf y R.L. London. "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generation." Comm. ACM 20 8 (Agosto 1977): 553-564.

(Siklóssy 1976)

L. Siklóssy. Let's Talk LISP. Englewood Cliffs, N.J.: Prentice-Hall, 1976.

(Snook y otros 1978)

T. Snook, C. Bass, J. Roberts, A. Nahapetian y M. Fay. Report on the Programming Language PLZ/SYS. New York: Springer Verlag, 1978.

(Steele 1975)

G.L. Steele. "Multiprocessing - Compactifying Garbage Collection." Comm. ACM 18 9 (Septiembre 1975): 495-508.

(Steele y Sussman 1980)

G.L. Steele y G.J. Sussman. "Design of a LISP-Based Microprocessor." Comm. ACM 23 11 (Noviembre 1980): 628-645.

(Tai 1980)

K.C. Tai. "Program Testing Complexity and Test Criteria." IEEE Transactions on Software Engineering SE-6 6 (Noviembre 1980): 531-538.

(Tanenbaum 1976)

A.S. Tanenbaum. "A Tutorial on ALGOL 68." ACM Surveys 8 2 (Junio 1976): 155-190.

(Tanenbaum 1978)

A.S. Tanenbaum. "A Comparison of PASCAL and ALGOL 68." Computer Journal 21 (1978): 316-323.

(Tausworthe 1977)

R.C. Tausworthe. Standardized Development of Computer Software. Englewood Cliffs, N.J.: Prentice-Hall, 1977.

(Teichrow y Hershey 1977)

D. Teichrow y E.A. Hershey III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing." IEEE Transactions on Software Engineering SE-3 1 (Enero 1977): 41-48.

(Teitelman 1975)

W. Teitelman. INTERLISP Reference Manual. Palo Alto, Calif.: Xerox Research Center Technical Report, 1975.

(Teitelman y Masinter 1981)

W. Teitelman y L. Masinter. "The INTERLISP Programming Environment." IEEE Computer 14 4 (Abril 1981): 25-33.

(Tennent 1973)

R.D. Tennent. "Mathematical Semantics of SNOBOL4." Proceedings ACM Symp. on Principles of Programming Languages. (Boston, 1973): 95-107.

(Tennent 1976)

R.D. Tennent. "The Denotational Semantics of Programming Languages." Comm. ACM 19 8 (Agosto 1976): 437-453.

(Tennent 1978)

R.D. Tennent. "Another Look at Type Compatibility in Pascal." Software-Practice and Experience 8 (1978): 429-437.

(Tichy 1979)

W.F. Tichy. "Software Development Based on System Structure Description." Proceedings 4th Conference on Software Engineering. IEEE cat. no. 79CH1479-5C (Munich, Septiembre 1979): 29-41.

(Valentine 1974)

S.H. Valentine. "Comparative Notes on ALGOL 68 and PL/I." Computer Journal 17 (1974): 325-331.

(van der Poel y Maarsden 1974)

W.L. van der Poel y L.A. Maarsden, editores. Machine Oriented Higher Level Languages. Amsterdam: North-Holland, 1974.

(van Wijngaarden y otros 1976)

A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, y R.G. Fisker. Revised Report on the Algorithm Language ALGOL 68. New York: Springer-Verlag, 1976.

(Walker y otros 1980)

B.J. Walker, R.A. Kemmerer y G.J. Popek. "Specification and Verification of the UCLA UNIX Security Kernel." Comm. ACM 23 2 (Febrero 1980): 118-131.

(Wegbreit 1974a)

B. Wegbreit. "The ECL Programming System." Fall Joint Comp. Conf. (1974): 253-262.

(Wegbreit 1974b)

B. Wegbreit. "The Treatment of Data Types in ELI." Com. ACM 17 5 (Mayo 1974): 251-264.

(Wegbreit y Spitzer 1977)

B. Wegbreit y J. Spitzer. "Programming Properties of Complex Data Structures." Journal ACM 23 2 (Abril 1976): 389-396.

(Wegner 1968)

P. Wegner. Programming Languages, Information Structures, and Machine Organization. New York: McGraw-Hill, 1968.

(Wegner 1972)

P. Wegner. "The Vienna Definition Language." ACM Computing Surveys 4 1 (1972): 5-63.

(Wegner 1976)

P. Wegner. "Programming Languages - The First 25 Years." IEEE Transaction on Computers C-25 12 (1976): 1207-1225.

(Wegner 1979)

P. Wegner, editor. *Research Directions in Software Technology*. Cambridge, Mass.: The MIT Press, 1979.

(Wegner 1980)

P. Wegner. *Programing with Ada: An introduction by Means of Graduated Examples*. Englewood Cliffs, N.J.: Prentice-Hall, 1980.

(Weinberg 1971)

G.M. Weinberg. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.

(Welsh y otros 1977)

J. Welsh, M.J. Sneedinger y C.A.R Hoare. "Ambiguities and Insecurities in Pascal." *Software - Practice and Experience* 7 6 (Noviembre 1977): 685-696.

(Wichmann 1973)

B.A. Wichmann. *ALGOL 60 Compilation and Assessment*. London: Academic Press, 1973.

(Williams 1979)

G. Williams. "Program Checking." *Proceedings Symposium on Compiler Construction - SIGPLAN Notices* 14 8 (Agosto 1979).

(Winograd 1979)

T. Winograd. "Beyond Programming Languages." *Comm. ACM* 22 7 (1979): 391-401.

(Wirth 1968)

N. Wirth. "PL360; A Programming Language for the 360 Computers." *Journal of the ACM* 15 1 (Enero 1968): 37-74.

(Wirth 1971a)

N. Wirth. "The Programming Language Pascal." *Acta Informatica* 1 (1971): 35-63.

(Wirth 1971b)

N. Wirth. "Program Development by Stepwise Refinement." *Comm. ACM* 14 4 (Abril 1971): 221-227.

(Wirth 1971c)

N. Wirth. "The Design of a Pascal Compiler". *Software - Practice and Experience* 1 (1971): 309-333.

(Wirth 1973)

N. Wirth. *Systematic Programming*. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

(Wirth 1974)

N. Wirth. "On the Composition of Wel-Structured Programs." *ACM Computing Surveys* 6 4 (Diciembre 1974): 247-259.

(Wirth 1975a)

N. Wirth. "On the Design of Programming Languages." En *Information Processing 74 (Proc. IFIP Congress 74)*. Amsterdam: North-Holland, 1975.

(Wirth 1975b)

N. Wirth. "An Assessment of the Programming Language Pascal." *IEEE Transactions on Software Engineering SE-1* 2 (Junio 1975): 192-198.

(Wirth 1976a)

N. Wirth. *Algorithms + Data Structures = Programs*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.

(Wirth 1976b)

N. Wirth. "Modula: A Language for Modular Multi-Programming." *Software - Practice and Experience* 7 (1977): 3-35.

(Wirth 1976c)

N. Wirth. "The Use of Modula." *Software - Practice and Experience* 7 (1977): 37-65.

(Wirth 1976d)

N. Wirth. "Design and Implementation of Modula." *Software - Practice and Experience* 7 (1977): 67-84.

(Wirth 1977)

N. Wirth. "Toward a Discipline of a Real-Time Processing." *Comm. ACM* 20 8 (Agosto 1977): 577-583.

(Wirth 1978)

N. Wirth. "Modula-2" Tech. Report 27. Zurich: Institut für Informatik, ETH, Diciembre 1978.

(Wirth 1979)

N. Wirth. "The Module: A System Structuring Facility in High-Level Programming Languages." Internal Tech. Report. Zurich: Institut für Informatik, ETH. Septiembre 1979.

(Wortman 1979)

D.B. Wortman. "On Legality Assertions on Euclid." *IEEE Transactions on Software Engineering SE-5* 4 (1979): 359-367.

(Wulf 1977)

Ver (Yeh 1977).

(Wulf y Shaw 1973)

W.A. Wulf y M. Shaw. "Global Variables Considered Harmful." *SIGPLAN Notices* 8 2 (Febrero 1973): 80-86.

(Wulf y otros 1971)

W.A. Wulf, D.B. Russell y A.N. Habermann. "BLISS: A Language for System Programming." *Comm. ACM* 14 12 (Diciembre 1971): 780-790.



(Wulf y otros 1972)

W.A. Wulf y otros. BLISS-11 Programmer's Manual. Maynard, Mass.: Digital Equipment Corp., 1972.

(Wulf y otros 1975)

W.A. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs y C.M. Gesche. The Design of an Optimizing Compiler. New York: American Elsevier, 1975.

(Wulf y otros 1976)

W.A. Wulf, R.L. London y M. Shaw. "An Introduction to the Construction and Verification of Alphard Programs." IEEE Transactions on Software Engineering SE-2 (Diciembre 1976): 253-265.

(Yeh 1977a)

R.T. Yeh, editor. Current Trends in Programming Methodology. Vol. 1, Software Specification and Design. Englewood Cliffs, N.J.: Prentice-Hall, 1977.

(Yeh 1977b)

R.T. Yeh, editor. Current Trends in Programming Methodology. Vol. 2, Program Validation. Englewood Cliffs, N.J.: Prentice-Hall, 1977.

(Yeh 1978)

R.T. Yeh, editor. Current Trends in Programming Methodology. Vol. 4, Data Structuring. Englewood Cliffs, N.J.: Prentice-Hall, 1978.

(Yourdon 1975)

E. Yourdon. Techniques of Program Structure and Design. Englewood Cliffs, N.J.: Prentice-Hall, 1975.

(Zahn 1974)

C.T. Zahn. "A Control Statement for Natural Top-Down Structured Programs." Symp. on Programming Languages. Paris, 1974.

(Zahn 1979)

C.T. Zahn. C Notes - A Guide to the C Programming Language. New York: Yourdon Press, 1979.

(Zelkowitz y otros 1979)

M.V. Zelkowitz, A.C. Shaw y J.D. Gannon. Principles of Software Engineering and Design. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

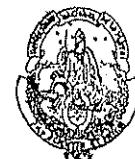
DONACION H.C. ZARVEL ZEGARRA

203
243
1353

\$.....

Fecha 16-5-05

Inv. E..... Inv. B. 1457



Donación Facultad

D.3
GHE

Fecha 21-10-2013

Inv. DIF-F057