UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Unidad Académica de los Grados Profesionales y de Posgrado del
Colegio de Ciencias y Humanidades
Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas

BIBLIOTECA DE COMPONENTES REUSABLES PARA TIPOS DE DATOS ABSTRACTOS IMPLEMENTADOS EN ADA

Tesis para la obtención del grado de
MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN
presentada por
Judith Leticia García González

1995

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

A mi madre: **María Luisa,**
A mi esposo: **Armando,**
A mis hijos: **Mónica,**
Armando y
Alberto

Por su apoyo, paciencia y consideración incondicional.

Quiero agradecer en forma muy especial a mi Directora de Tesis Dra. Hanna Oktaba por su apoyo invaluable para la realización de este trabajo.

Asimismo agradezco los comentarios y observaciones siempre acertados de mis sinodales:

Mat. Carlos Velarde Velázquez,
M. en C. Guadalupe Ibargüengoitia González,
M. en C. Mónica Ardisson Pérez y
Dr. Marcelo Mejía Olvera

ÍNDICE GENERAL

INTRODUCCIÓN

El fenómeno conocido como crisis de *software*, surgido a finales de la década de los sesenta, ha influido en la búsqueda de principios y características que permitan obtener software de calidad, sobre todo en sistemas complejos. Dentro de los principios aceptados se encuentra el concepto de reusabilidad, propuesto hace ya más de 20 años por McIlroy, que, aunque aceptado, no ha logrado vencer los obstáculos, principalmente técnicos, para alcanzar el avance espectacular que se esperaba.

Dentro de las técnicas de reusabilidad existentes, llaman la atención aquellas que se basan en composición y que están fundamentadas en la idea de ensamblar componentes, con ninguna o escasa modificación, para crear el sistema de *software* deseado.

Naturalmente, los componentes para ser reusables deben presentar determinadas características, principalmente tener una definición precisa y fácil de entender, ser confiables, flexibles y fáciles de componer.

En la actualidad, dentro de las nuevas metodologías para el desarrollo de grandes sistemas de *software*, destaca el paradigma orientado a objetos que ha aportado una nueva filosofía con grandes posibilidades de aplicación, no sólo en la etapa de implementación, a través de los lenguajes orientados a -o basados en- objetos, sino también en las etapas de diseño y mantenimiento, promoviendo la modularidad y el reuso de componentes.

Por otro lado, las ideas relacionadas con los tipos de datos abstractos permiten definir en forma abstracta y precisa los datos y las operaciones que se van a manejar, sin necesidad de detallar la forma en que serán implementados.

Las especificaciones formales suplen muchas de las deficiencias que tienen las especificaciones informales; dentro de ellas, las especificaciones algebraicas han logrado destacar como un instrumento formal muy útil que respalda los esfuerzos tendientes a lograr *software* eficiente.

Las especificaciones abstractas, por una parte, y las metodologías para la creación de módulos reusables, por otra, facilitan la implementación de diseños basados en objetos.

Ada es un lenguaje moderno, con un potencial muy amplio y una difusión creciente, que tiene características que lo llevan a ser considerado dentro de los lenguajes basados en objetos y que ofrece estructuras, como los paquetes y los subprogramas genéricos que permiten realizar -con cierta facilidad- la implementación de tipos de datos abstractos a la manera de módulos parametrizables .

Con base en estas ideas, surgió la idea de desarrollar una biblioteca de componentes reusables de tipos de datos abstractos con cuyas especificaciones algebraicas ya se contaba, que fuera implementada en Ada.

La biblioteca constituiría el inicio, en este lenguaje de programación, de un conjunto creciente de módulos eficientes, confiables, flexibles, basados en especificaciones formales, fáciles de componer, que permitieran el desarrollo de proyectos con alcances cada vez mayores.

El antecedente directo de esta tesis es el trabajo desarrollado por la Dra. Hanna Oktaba referente a componentes reusables basados en especificaciones algebraicas implementados en Modula-2. Sus conceptos han impulsado el desarrollo de otras dos tesis en la Maestría de Ciencias de la Computación: una biblioteca de componentes reusables basados en las mismas especificaciones algebraicas pero implementados en el lenguaje ML, desarrollada por Guadalupe Ibargüengoitia y un sistema de clasificación para bibliotecas de componentes reusables basados en especificaciones algebraicas, pertenecientes a implementaciones diferentes, desarrollado por Armando Hernández.

En el capítulo uno se habla sobre conceptos generales de reusabilidad, los tipos de reusabilidad que se han venido dando y, dentro de ellos, los sistemas basados en composición de módulos equiparables a *chips* de *hardware*.

Los tipos de datos abstractos y las especificaciones formales se tratan en el capítulo dos. En forma particular se mencionan las especificaciones algebraicas y se detalla un caso ilustrativo en lenguaje LESPAL, que es el lenguaje empleado en las especificaciones algebraicas adoptadas.

En el capítulo tres se definen las características ideales de un lenguaje de programación, necesarias para implementar especificaciones formales y se confrontan con las estructuras disponibles en Ada, que es el lenguaje que se ha seleccionado para llevar a cabo la programación de los módulos.

El capítulo cuatro explica las decisiones generales tomadas en relación con la implementación desarrollada.

Se presenta un caso ilustrativo con la reseña de decisiones específicas relacionadas con la implementación y la validación realizada en el capítulo 5.

Por último, se presentan las conclusiones.

Las especificaciones algebraicas y el código de los tipos de datos abstractos implementados se incluyen en el Anexo A.

El Anexo B contiene el código completo del programa de validación de la implementación correspondiente al ejemplo de colas con prioridades.

1. COMPONENTE REUSABLE COMO CONCEPTO DE INGENIERÍA DE SOFTWARE

La Ingeniería de Software[1], disciplina joven que surgió como una respuesta a la llamada "crisis de software", consiste en la aplicación de conocimientos científicos en el diseño y construcción de programas de computación y en la documentación correspondiente requerida para desarrollar, operar y darles mantenimiento.

Pressman, Meyer y otros autores[2] están de acuerdo en que la meta más importante de la **Ingeniería de Software** es la de ayudar a producir *software* de calidad.

La calidad es un concepto subjetivo, difícil de definir. Sin embargo, para tener una idea de lo que se quiere decir con calidad del software se citará a Pressman, para quien la calidad puede definirse de la siguiente manera[3]:

> "Concordancia con los requerimientos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo *software* desarrollado profesionalmente."

Dentro de las cualidades deseables que se esperan de un software de calidad están la correctez, confiabilidad, robustez, eficiencia, así como la posibilidad de que sea verificable, mantenible, transportable, comprensible, amigable con el usuario, entre otras.

En ingeniería, tradicionalmente se han considerado como ideas fundamentales la acumulación de experiencia en el diseño, la fijación de normas para estandarizar requerimientos, el establecimiento de interfaces bien definidas para facilitar los intercambios y reducir la complejidad de una solución.

---

[1] (BOEH079 cit en [SOMM87])
[2] [PRES809], [MEYER88]
[3] [PRES809]

Algunas de estas ideas corresponden a los conceptos que ahora se manejan en relación con la reusabilidad y, especialmente, con la reusabilidad del *software*, que es el tema de este capítulo.

1.1 REUSABILIDAD DEL SOFTWARE

Hasta hace muy poco, apareció por primera vez, en un diccionario en inglés, la definición de la palabra "reuso". En español, no aparecen ni "reuso" ni "reutilización", que podrían ser los términos equivalentes y que, de hecho, son los términos empleados en la escasa literatura correspondiente disponible. En este trabajo se empleará el término "**reuso**" y sus derivados para denotar la repetición del uso.[4]

En el área de la computación, el interés en el *software* **reusable** surge cuando los programadores se dan cuenta de que una forma de incrementar la productividad durante la creación de un sistema particular es escribir menor cantidad de *software*, siempre y cuando se mantenga el mismo grado de funcionalidad.[5]

Ya en 1950 Wilkes, Wheeler y Gill hicieron notar la importancia de las bibliotecas de subprogramas. En 1969 McIlroy replanteó esta idea al proponer una tecnología de componentes de *software* **reusables** a la manera de las tecnologías de componentes para automóviles:[6]

> "Me gustaría que el estudio de los componentes de *software* llegara a ser una rama digna de la Ingeniería de *Software*. Me gustaría ver catálogos estandarizados de rutinas clasificadas por precisión, robustez, requerimientos de tiempo y espacio y tiempo de ligado de parámetros."

Una mayor preocupación por desarrollar una tecnología viable de componentes de *software* impulsó, en los años setenta, la investigación en abstracción de datos, programación orientada a objetos y modularidad, todos ellos conceptos muy apreciados en Ingeniería de *Software*.

Posteriormente, en los años ochenta se vislumbraron perspectivas interesantes para los componentes de *software* gracias a la investigación de abstracción de procesos y computación distribuida.

Pero, ¿en qué consisten el reuso y la reusabilidad?

Veamos lo que diversos autores dicen al respecto:

Reuso es la utilización, en condiciones diferentes, de conceptos previamente adquiridos. Por otro lado, **reusabilidad** es la medida de la facilidad con la cual uno puede aplicar esos conceptos u objetos previos en una nueva situación.[7]

---

[4] ("re" - prefijo latino que forma parte de varias voces de nuestra lengua e indica repetición
+ "uso" - explotación o aprovechamiento de algo).
[5] [NEIGH89]
[6] [WEGNE89]
[7] [PRIET89]

Wegner define a la **"reusabilidad"** como un principio general de ingeniería cuya importancia se deriva del deseo de evitar duplicación y de capturar los rasgos comunes en clases de tareas intrínsecamente parecidas.[8]

Para Prieto-Díaz, el **reuso de *software*** consiste en la utilización de componentes de *software* existentes para desarrollar nuevos sistemas.[9]

Biggerstaff y Perlis definen el **reuso de *software*** , en forma muy amplia, como "la reaplicación de una variedad de tipos de conocimiento de un sistema hacia otro parecido, para reducir el esfuerzo de desarrollo y mantenimiento de ese otro sistema. Este conocimiento reusado incluye elementos tales como: conocimiento del dominio, experiencia de desarrollo, decisiones de diseño, estructuras de su arquitectura, requerimientos, diseños, código, documentación, etc."[10]

La idea subyacente en estas definiciones consiste en crear sobre algo sólido ya existente, probado, confiable, y no "inventar la rueda" cada vez que se necesite; aprovechar el conocimiento acumulado para incrementar la productividad y mejorar la calidad de los productos resultantes.

Las soluciones ya probadas que se usan una y otra vez para resolver el mismo tipo de problema llegan a ser aceptadas, generalizadas y estandarizadas.[11]

La frase que establece que nos deberíamos apoyar en los hombros de unos y otros, más que en los pies de cada uno podría interpretarse como un argumento en favor de la reusabilidad tanto material como intelectual.

La reusabilidad proporciona tanto una justificación científica por la investigación que simplifica y unifica clases de fenómenos como una justificación económica para desarrollar productos de *software* reusable que hagan a los programadores y a las computadoras más productivas.

1.2 TIPOS DE REUSABILIDAD

Al hablar de Al hablar de reusabilidad, con frecuencia se hace referencia a muy diversas formas en las que ésta se puede dar, las cuales pueden ir desde el reuso de una especificación (a través de especificaciones formales), pasar por el reuso de diseños y código, seguir con el reuso de generadores de aplicaciones, hasta llegar al reuso de paquetes comerciales listos para instalarse.[12]

Desde el punto de vista de Biggerstaff y Perlis[13], las tecnologías que se aplican al problema de la reusabilidad se pueden dividir en dos grandes grupos, de acuerdo con la

---

8 [WEGNER89]
9 [PRIET91]
10 [BIGGERS89a]
11 [PRIET91]
12 [BIGGERS89b], [NEIGHB89], [PRATH89]
13 [BIGGERS89b]

naturaleza de sus componentes: tecnologías de composición y tecnologías de generación. Esta distinción se puede hacer extensiva para clasificar a los sistemas de *software* resultantes.

1.21 Sistemas basados en composición

Cuando se trata de un sistema complejo es difícil entenderlo si se toma como un bloque único. Resulta mucho mejor si se le descompone en partes más pequeñas que tengan una relación lógica, es decir, unidades coherentes a las que se acostumbra llamar "componentes" o "módulos".

Esas unidades o módulos se consideran como "cajas negras" que pueden ser complejas en su interior pero externamente se tratan como elementos simples que facilitan la composición y mantenimiento de un sistema grande.

En las teorías de composición, las partes que se van a reusar son atómicas y, en principio, no serán cambiadas en el transcurso de su reuso. Bajo esta concepción, la creación de nuevos programas es únicamente un problema de composición.

Por supuesto, los componentes pueden ser modificados o cambiados para que se ajusten mejor a los propósitos de quien los va a reusar.

Ejemplos de este tipo son: esqueletos de código, subrutinas, funciones, programas y objetos (al estilo de los lenguajes de programación orientados a objetos).[14]

En este modelo hay quien se centra en proporcionar al usuario una interfaz para un sistema de composición que verdaderamente emplee a los componentes como "cajas negras". Permiten que el usuario seleccione las operaciones abstractas a realizar y los tipos abstractos en los que esas computaciones operarán, pero es el sistema el que dará el régimen de interconexión.

Asimismo, hay otros modelos que se enfocan en desarrollar programas a partir de especificaciones formales.

1.22 Sistemas basados en generación

Los sistemas basados en generación se logran reusando patrones que conducen a la creación de versiones adaptadas de sí mismos. Los patrones son las semillas de las cuales se obtendrán nuevos componentes especializados.[15]

Los componentes que se reusan no son entidades concretas sino patrones incorporados en la manufactura de un programa generador. Las estructuras resultantes guardan un ligero parecido con los patrones de los programas que los generaron y, muy posiblemente, tendrán poco parecido con otras instancias generadas a partir del mismo código patrón.[16]

---

[14] [BIGGER89b]
[15] [BIGGER89c]
[16] [BIGGER89b]

En este modelo, los autores distinguen entre sistemas basados en un lenguaje, generadores de aplicaciones y sistemas basados en transformaciones; sin embargo, reconocen que muchos de los casos de estudio presentados tienen características que los clasificarían en más de un grupo.

1.3 COMPONENTES REUSABLES EQUIPARABLES A CHIPS DE HARDWARE

El modelo de reuso basado en la composición se fundamenta en la idea de ensamblar componentes, con o escasa modificación, para crear el sistema de software deseado. Este modelo es el equivalente a la idea de hardware de enchufar chips de circuitos integrados para desarrollar sistemas de hardware.

Los grandes sistemas que son fácilmente mantenidos y apreciados tienen la propiedad de ser modulares y construibles de una manera gradual a partir de componentes primitivos.

Wegner[17] dice que los componentes de *software* son bloques de construcción, que implican una inversión fuerte de capital, con los cuales se construyen programas de grandes dimensiones y que el papel de las bibliotecas de componentes es el de un almacén de conocimiento que organiza la interacción de los componentes de *software*, tanto durante el desarrollo como durante la ejecución del programa.

1.31 Características de los componentes

Para que los módulos o componentes tengan un potencial aceptable de reusabilidad deben cumplir con criterios como los siguientes :

- deben ser el reflejo de un buen nivel de abstracción, lo que les otorgará claridad para que se comprenda su funcionamiento, se tenga confianza en su efectividad y se facilite su manejo;
- deben ser flexibles es decir, deben estar diseñados de tal forma que se les pueda combinar facilitando la composición de un gran sistema;
- deben permitir una composición ortogonal, esto es, los módulos deben ser independientes del contexto que los incluye;
- deben ser independientes de un ambiente específico de ejecución[18];
- deben ser eficientes.[19]

Más allá de los criterios anteriormente citados, la característica básica de un módulo reusable es la de que realmente realice la acción que se espera de él, esto es, su efectividad.

En principio los componentes con un alto potencial de reuso no deben requerir modificación y se puede esperar que tengan una aplicación amplia.

---

[17] [WEGNER89]
[18] Estas dos últimas características permitirán que un componente se pueda transportar con facilidad.
[19] [GARGAR7]

1.32 Limitaciones en la reusabilidad de componentes

Es una creencia difundida el hecho que el reuso sea la clave para mejorar la productividad y la calidad del desarrollo de *software*.

Cuando se requiere el diseño de grandes sistemas distribuidos, asíncronos de gran complejidad, no se tiene más opción que diseñar con base en el conocimiento acumulado a través del reuso.

El reuso de componentes de *software* aumenta las capacidades del desarrollador de *software*, permitiéndole escribir menos símbolos en total para el desarrollo del sistema y emplear menos tiempo en el proceso de organizar esos símbolos. Pero a pesar de la gran promesa que representa, ésta no se ha podido alcanzar del todo.

Las razones que han frenado el desarrollo de componentes de software reusable aplicables a sistemas generales pueden resumirse en: costo inicial fuerte requerido, cierta indiferencia de parte de los usuarios potenciales y, sobre todo, falta de una base tecnológica adecuada que incluya, entre otras cosas, un buen número de herramientas básicas y un amplio catálogo de componentes reusables listos para ser operados.

Hasta que se cuente con un ambiente poblado de herramientas y de componentes se podrá explotar verdaderamente el reuso.

1.33 Problemas tecnológicos

Los métodos para definir componentes son imprecisos; es difícil describir un componente a un usuario; no hay herramientas para crear, clasificar, refinar y combinar componentes de una forma eficiente.

Por ello, para que el concepto de *software* reusable mediante componentes llegue a ser una realidad, se deben resolver adecuadamente problemas entre los que destacan dos grupos que se describirán a continuación.

1.331 Problemas relacionados con la naturaleza de los componentes[20]

a) Mecanismos para identificar componentes.

Debido a que se invierte mucho más tiempo en determinar requerimientos, hacer el diseño y dar mantenimiento, que en la codificación, es necesario encontrar maneras para determinar qué componentes son por lo general útiles y adaptables a diferentes proyectos.

Sin embargo, el esfuerzo hecho para identificar tales componentes sin restringirse a un dominio específico no parece una tarea fácil.

---

[20] [HOROW87]

b) Métodos para especificar componentes.

Definir formas de describir un componente funcional, una vez que éste ha sido seleccionado, de tal manera que otros programadores puedan entenderla.

Se puede considerar cualquiera de los diversos formalismos de especificación conocidos, aunque aceptando de antemano que éstos no son completamente inteligibles para todas las personas.

c) Forma de los componentes.

Decidir si serán implementados en algún lenguaje de programación o descritos por un lenguaje de diseño de programas que permita intercalar estructuras de programación con el lenguaje de programación preferido.

d) Catálogos de componentes

Determinar cómo van a ser catalogados los componentes. Puesto que van a ser muchos, se necesita contar con un lenguaje de meta-descripción para agrupar componentes parecidos.

1.332 Problemas relacionados con el funcionamiento[21]

Para que el reuso pueda funcionar adecuadamente se deben resolver los siguientes problemas: localización, comprensión, modificación y composición de módulos.

a) Localización

La localización implica tanto la selección de componentes como su concordancia. Incluye ubicar componentes que sean muy parecidos, que aún cuando requieran ser parcialmente modificados, puedan adaptarse con un menor esfuerzo a las especificaciones requeridas. Esto, básicamente, significa tener componentes organizados en bibliotecas con un esquema adecuado de clasificación.

b) Comprensión

Para que un usuario pueda aprovechar un módulo en forma adecuada tiene que entenderlo. Sobre todo si el componente va a ser modificado, el usuario necesita tener acceso a información adicional detallada que le permita modificar el módulo sin afectar la esencia de su funcionamiento.

c) Modificación

El proceso de modificación es el alma de la reusabilidad. Consiste en convertir una biblioteca de unidades estáticas en un sistema de componentes dinámicos que evolucionen conforme cambien los requerimientos del sistema.

---

[21] [illegible] y [illegible]

Sin embargo, la modificación o extensión de un componente puede requerir conocimiento más detallado acerca de la implementación del mismo, negando así muchas de sus ventajas. Si esos cambios se operan sobre diversos componentes reusables, sus virtudes se verán seriamente disminuidas.

d) Composición

El proceso de composición impone requisitos difíciles a las representaciones usadas para especificar componentes ya que, por una parte, deben ser estructuras compuestas que se comporten como entidades independientes con características computacionales bien definidas y que, a la vez, permitan componerlas en nuevas estructuras de cómputo con características computacionales diferentes, metas que podrían parecer antagónicas.

Por otro lado, si el trabajo requerido para combinar componentes lleva mucho tiempo o requiere gran sofisticación, las ventajas potenciales del código reusable se verán disminuidas en gran medida. Los grandes sistemas de *software* deben ser construidos de tal manera que la modificación y evolución puedan ser realizadas en un tiempo proporcional a la magnitud de los cambios más que al tamaño del sistema.

Para Prieto-Díaz[22], los conceptos de modificación y composición quedan incluidos en lo que él llama adaptación que depende, por una parte, de las diferencias entre los requerimientos planteados y las características ofrecidas por los componentes existentes y, por otra, de la habilidad del "reusador".

1.34 Bibliotecas de componentes reusables

Las bibliotecas promueven el reuso del conocimiento existente en la creación de nuevo conocimiento.

El diseño de bibliotecas está relacionado con la organización de colecciones de componentes de *software* a fin de que puedan ser fácilmente creados y usados.

Las bibliotecas de programas deben contener no sólo componentes de *software* sino también herramientas para organizar el conocimiento, tales como catálogos. En este aspecto son como bibliotecas de libros o filmotecas. Sin embargo difieren de las bibliotecas convencionales en que sus clientes pueden ser tanto computadoras como seres humanos, por lo mismo, las herramientas de biblioteca para ensamblar componentes en programas deberán ser automáticas.

A la fecha ya se tiene experiencia con dos tipos de bibliotecas de subrutinas que podrían considerarse como componentes reusables: las bibliotecas proporcionadas por los proveedores de *software* que contienen un conjunto de funciones o procedimientos de utilidad general para todos los usuarios y las bibliotecas de subrutinas aplicables a áreas específicas, como las bibliotecas estadísticas (BMD o SPSS) o las bibliotecas especializadas en análisis numérico (IMSL), entre otras.

---

[22] [PRIET87]

Los dos casos mencionados tienen varios puntos en común, los cuales han influido en su éxito:

- ◊ **el propósito de cada módulo se puede definir, por lo general, de una forma concreta en el idioma cotidiano;**
- ◊ **cada rutina es fija excepto por algunos parámetros, los cuales afectan su operación de una manera bien definida;**
- ◊ las rutinas pueden ser intercaladas en un programa de usuario y usadas por ese programa, independientemente del lenguaje que se esté empleando.

Por lo que toca a las bibliotecas de subrutinas mencionadas, la definición, clasificación e interfaz de sus elementos es una tarea sin complicaciones.[23]

La idea de contar con componentes de código reusable surge del deseo de incrementar el éxito de las bibliotecas de subrutinas.

Wegner[24] considera que al diseñar una biblioteca de programas se deben considerar las siguientes cuestiones:

- ⇒ clases de componentes que la biblioteca puede contener;
- ⇒ granularidad y dominio de aplicación de la biblioteca;
- ⇒ clases de clientes (programas, personas) que usarán la biblioteca;
- ⇒ forma en que los componentes se cargan, ligan e invocan;
- ⇒ forma en que los componentes son creados, insertados, inspeccionados y recuperados;
- ⇒ relaciones entre componentes que pueden expresarse; y
- ⇒ tipo de conocimiento que se requiere para ayudar a los programadores a construir programas compuestos a partir de bibliotecas de componentes de *software*.

[23] [HOROWI87]
[24] [WEGNER89]

2. TIPOS DE DATOS ABSTRACTOS Y ESPECIFICACIONES ALGEBRAICAS

El objetivo de esta tesis es implementar una biblioteca de componentes reusables basados en especificaciones algebraicas de tipos de datos abstractos. Pero ¿qué son las especificaciones y qué son los tipos de datos abstractos?

Los datos tienden a ser la parte estable de un sistema. Por ello, cuando se habla de técnicas para construir software de calidad, es frecuente encontrar ejemplos en la literatura en torno a los datos, más que en torno a las funciones. El estructurar en base a los datos protege al sistema con un grado más alto de continuidad y reusabilidad.

De esta manera, se puede observar que el punto clave en el diseño de sistemas de software llega a ser la búsqueda de tipos de datos abstractos.

En este capítulo se tratará de aclarar lo que se entiende por tipos de datos abstractos y sus especificaciones algebraicas y se presentará un caso ilustrativo, el de colas con prioridades, cuya implementación también se comentará detalladamente en el capítulo cuatro.

2.1 TIPOS DE DATOS ABSTRACTOS

"Tipo de datos" es el término usado por los diseñadores de lenguajes de programación para denotar al conjunto de valores que una variable puede tomar.

Se da el nombre de **"tipos de datos abstractos" (TDA)** a las definiciones hechas en forma abstracta y precisa de un tipo de datos, junto con el grupo de operaciones válidas para ese tipo, sin que sea necesario conocer su representación interna.

Una abstracción implica la consideración únicamente de los aspectos importantes, dejando de lado los detalles poco relevantes. Además las abstracciones se pueden ver a diferentes niveles. Conforme se pasa a grados más altos de abstracción se logra una visión más amplia, en tanto que los detalles de niveles inferiores quedan ocultos.

Existen dos niveles conceptuales diferentes que se reflejan en el diseño y facilitan la implementación de un sistema. Corresponden a dos enfoques distintos resultantes de la aplicación del punto de vista de un usuario o de un programador:

- el usuario los identifica por su tipo y el programador por su estructura;
- al usuario le interesa conocer las operaciones disponibles para un tipo de datos y al programador le interesa la estructura de la memoria que puede emplear para implementar estas operaciones de manera factible y eficiente.

Los **TDA** son clases de objetos caracterizados por un conjunto de operaciones que manipulan sus datos, pero haciendo abstracción de la forma en que dichas operaciones pueden implementarse y de las estructuras de datos que se pueden utilizar para lograrlo.

2.2 ESPECIFICACIONES

2.21 Definición

En la búsqueda de métodos de ingeniería que puedan ser aplicados al diseño y desarrollo de software se ha determinado que, al menos, éste debe contar con las propiedades de corrección y flexibilidad que garanticen la posibilidad de adaptación a las necesidades cambiantes de su ambiente.

Sin embargo, para medir su corrección es necesario poderlo comparar con algún comportamiento ya establecido y que esté descrito en una especificación.

En forma general, se puede decir que una **especificación** es la descripción de una entidad en la que se establecen sus propiedades esenciales. Las descripciones pueden ser orales o escritas, pero sólo se les llama especificaciones cuando se les captura como base de un trabajo posterior.[1]

Tratándose de software se habla de la especificación en sus diferentes etapas de su desarrollo. De esta manera se puede tener:

- Especificación de requerimientos, que define las necesidades planteadas por el usuario;
- especificación del diseño, que define la manera en que se cubrirán esas necesidades;

[1] [GOLDS85] 2.

- especificación de la implementación, que define una solución particular al problema de construcción del sistema que fue definido en la especificación de diseño.

En particular para esta tesis, las especificaciones que resultan más interesantes son las referentes al diseño, ya que las especificaciones algebraicas de **TDA** en las que se basa la implementación desarrollada son de este tipo.

2.22 Características

La especificación deberá cumplir todos los lineamientos marcados por los principios de Ingeniería de Software (abstracción, ocultamiento de la información, modularidad, uniformidad, completez).

Es especialmente importante que reúna las características de ser clara, comprensible, precisa, consistente y de ser suficientemente completa.

2.23 Papel que desempeñan

Las especificaciones desempeñan diversas funciones, entre las que se tienen por ejemplo:

- al definir las necesidades del usuario del producto, establecen un compromiso entre éste y el diseñador;
- al definir los requerimientos de la implementación, aclaran y definen un compromiso entre los diseñadores y los implementadores;
- sirven como un mecanismo de referencia y de apoyo de las decisiones de diseño e implementación tomadas;
- tratándose de especificaciones formales, constituyen el punto de partida para verificaciones y validaciones;
- sirven de referencia durante la fase de mantenimiento.

2.24 Clasificación

En la actualidad existe un consenso respecto a que, en los proyectos complejos de *software*, es imperativo contar con especificaciones adecuadas para obtener calidad.

Las especificaciones informales expresadas en lenguaje natural por sí solas no son apropiadas porque resultan extensas, incompletas, inconsistentes, inexactas y ambiguas. Sin embargo, pueden ser de utilidad en una primera etapa de introducción al sistema de software y como complemento para asegurar la legibilidad de las especificaciones formales.

Se habla de una especificación formal cuando la especificación se expresa a través de un sistema de notación con sintaxis y semántica precisas que impiden crear expresiones ambiguas.

Las especificaciones formales pueden ser de distintos tipos, por ejemplo[2]:

- Lógicas, las cuales están basadas en el sistema formal de la lógica de primer orden. Consisten en la definición de predicados que describen las condiciones antes y después

[2] [OKTABA88], [GUEZZ91]

de la ejecución de una proposición. Si la precondición es verdadera antes de ejecutar una determinada proposición y la ejecución de la misma termina, se concluye que la postcondición también debe ser verdadera. Fue descrito por Hoare en 1969[3].

- Funcionales, que se basan en la semántica propia de cada lenguaje de programación. Sólo indican las funciones que deben ejecutar los programas, ya que a éstos los considera como funciones que transforman valores de entrada en valores de salida.
- Algebraicas, basadas en los tipos de datos abstractos a los que se conoce como álgebras. En la especificación se describen las operaciones, los dominios y codominios sobre los que están definidas y se proporciona un conjunto de axiomas que describe el comportamiento de dichas operaciones.

2.25 Ventajas y limitaciones de las especificaciones formales

Las ventajas que se mencionan más frecuentemente para apoyar el uso de las especificaciones formales son[4]:

- Las especificaciones realizadas mediante lenguajes formales, generalmente, son más concisas y cortas que las hechas utilizando lenguaje natural;
- permiten que el diseñador use razonamiento con rigor matemático; las propiedades de una especificación pueden probarse de la misma manera en que se prueban los teoremas matemáticos;
- se produce un software de mejor calidad, ya que corresponde a los requisitos solicitados y contiene menor cantidad de errores;
- se logra mayor productividad, ya que los errores, al ser eliminados en una etapa temprana del desarrollo resultan menos costosos;
- permiten su adaptación para automatizar la producción de herramientas necesarias durante el formateo, análisis, modificación y, sobre todo, verificación, para ser usadas a lo largo de toda la etapa del desarrollo.

Dentro de las limitaciones de los métodos formales están:

- ser incomprensibles para la mayoría de los usuarios finales y de los clientes, lo cual es una desventaja porque son precisamente los clientes los que deben validar las especificaciones funcionales;
- todavía no existen técnicas formales disponibles para todas las áreas de especificación;
- no hay herramientas que den soporte adecuado a la construcción de las especificaciones formales;
- hasta el momento, únicamente han sido usadas como elementos de investigación y, aunque existen reportes de aplicaciones prácticas en la industria, su uso aún está limitado.

---

[3] [HOARE69]
[4] [GOLD885] y [VANH089]

2.3 ESPECIFICACIONES ALGEBRAICAS

Para lograr la expresión de propiedades de un sistema se han propuesto muchos formalismos matemáticos. Entre ellos se cuenta con un estilo descriptivo, bastante difundido, basado en el uso del álgebra como formalismo matemático subyacente.

En esencia, las especificaciones algebraicas describen un sistema como una álgebra heterogénea, es decir, como una colección de conjuntos diferentes, sobre la cual se definen diversas operaciones. Este concepto se acerca mucho a la idea de tipos de datos abstractos que es básica en los sistemas de software y de la cual ya se habló en la primera parte de este capítulo.

Bajo esta perspectiva, se puede decir que una especificación algebraica[5] es la descripción matemática de un tipo de datos abstracto.[6]

Como tienen un fundamento matemático riguroso, a las especificaciones algebraicas se les puede dar un significado bien definido, independiente de la implementación.

2.31 Construcción

Al construir una especificación, primero se debe lograr la abstracción de las propiedades y operaciones que tiene el tipo de datos que se va a describir.

Posteriormente, se definen las funciones básicas aplicables al tipo de datos, señalando el número y tipo de parámetros requeridos y se precisa su significado mediante axiomas.

Como culminación de la fase de construcción está la verificación. Tal proceso consiste en probar mediante métodos matemáticos que la especificación algebraica construida es precisa, suficientemente completa y consistente.[7]

2.32 Lenguajes de especificación

Ya en el inciso anterior se indicó la posibilidad de emplear el lenguaje natural para crear especificaciones; sin embargo, también se mencionaron sus inconvenientes y deficiencias así como la necesidad de contar con un lenguaje formal que consistiese en un sistema de notación con una semántica precisa que le impida crear expresiones ambiguas.

Por ello, en los años sesenta se intensificó la búsqueda de lenguajes con suficiente poder de abstracción que facilitaran el diseño y la verificación de programas. Se exploraron, sobre todo, distintos métodos matemáticos. Como las fórmulas matemáticas tienen una sintaxis y semántica precisas llegan a ser una manera natural de expresar las propiedades de un sistema.

Para la especificación de los TDA, los lenguajes que mejor acogida tuvieron fueron aquellos basados en los métodos algebraicos porque se observó que éstos permitían expresar, en forma abstracta, conjuntos de datos junto con las operaciones que los manipulan.

Para poder definir un TDA se requiere de un lenguaje que permita expresar los siguientes aspectos[8] :

---

5 [MARTI86]
6 [VANHO89]
7 [OKTAB88]
8 [OKTAB90]

- La descripción sintáctica del tipo y sus operaciones,
- la descripción abstracta de la semántica de las operaciones,
- modularización,
- encapsulación,
- composición y extensión de cápsulas, y
- parametrización de cápsulas.

Los primeros lenguajes que ofrecieron características que facilitaban las especificaciones fueron CLEAR (el cual trata, entre otras cosas, la parametrización), Larch, ACT ONE y OBJ2.[9] Las especificaciones algebraicas que se utilizan en este trabajo fueron hechas en LESPAL, lenguaje en español, basado principalmente en conceptos de Van Horebeek[10].

2.4 Un caso ilustrativo: Colas con prioridades

A continuación se presentará el esquema de la especificación algebraica del tipo de datos abstracto denominado **colas con prioridades**, para ilustrar la manera como el lenguaje LESPAL permite expresar los elementos citados en el punto anterior.

2.41 Definición

Una cola es una secuencia lineal de un número arbitrario de elementos del mismo tipo en la cual éstos son incorporados por un extremo y removidos por el otro.

En la cola con prioridades, la prioridad de un elemento afecta el orden en el cual éste será atendido; entre mayor sea su prioridad más pronto será retirado de la cola.

En la vida diaria existen muchas aplicaciones para este tipo como la asignación de pista a aviones con problemas de combustible o de funcionamiento, o la asignación de recursos que realiza el sistema operativo de un sistema multiusuario respecto a impresoras, accesos a disco, tiempo de procesador, etcétera, a usuarios que ostentan distintos grados de preferencia de atención.

2.42 Esquema de la especificación algebraica

La especificación algebraica normal aparece en el Anexo A. Aquí se le presenta separada en secciones para destacar los elementos más relevantes.

---

[9] [VANHO89]
[10] [OKTA89b]

ESQUEMA ColasConPrioridadesEsq

Parámetros:

```
PARAM ElementosOrd;
   IMPORTA Bool DE Booleanos;
   EXPORTA TODO;
   GENERO Elem;
   OPERACIONES
     _=_:Elem*Elem → Bool;
     _<_: Elem*Elem → Bool;
   VAR e1,e2,e3:Elem;
   AXIOMAS
     (e1=e1)=cierto
     (e1=e2)=(e2=e1)
     (e1=e2) y (e2=e3) ⇒ (e1=e3)=cierto
     (e1=e2) ⇒ (e1<e2)=falso
     (e1<e2) ⇒ (e2<e1)=falso
     (e1<e2) y (e2<e3) ⇒ (e1<e3)=cierto
FIN DE PARAM ElementosOrd;
```

Especificación:

```
ESPEC ColasConPrioridades;
   IMPORTA TODO DE ElementosOrd
   EXPORTA TODO;
   GENERO ColaPr;
   OPERACIONES
     [OP1] vacia: → ColaPr;
     [OP2] mete:Elem*ColaPr → ColaPr;
     [OP3] sacamax:ColaPr → ColaPr;
     [OP4] maximo:ColaPr → Elem;
     [OP5] esvacia:ColaPr → Bool;
   VAR c:ColaPr, e:Elem;
   AXIOMAS
     [CP1] sacamax(vacia)=error
     [CP2] sacamax(mete(e,c))= SI esvacia(c) ENT vacia
                                  SI_NO SI maximo(c)< e ENT c
                                        SI_NO mete(e,saca max(c))
     [CP3] maximo(vacia)=error
     [CP4] maximo(mete(e,c))=SI esvacia(c) o maximo(c)< e ENT e
                                  SI_NO maximo(c)
     [CP5] esvacia(vacia)=cierto
     [CP6] esvacia(mete(e,c))=falso
FIN DE ESPEC ColasConPrioridades;
```

FIN DE ESQ ColasConPrioridadesEsq

2.43 Detalles del esquema de especificación

Debido a que es un esquema, se trata de un patrón general de colas con prioridades, parametrizado por tipo de elementos. Sin embargo, para que pueda efectivamente aplicarse, es necesario que se haga una instanciación referida a un tipo de datos específico.

El esquema consta de dos partes básicas:

- Los **parámetros** que presentan el género de los tipos (con las operaciones aplicables y los axiomas que deben cumplir esas operaciones, así como la mención de las relaciones que se establecen, a través de sus importaciones y exportaciones) necesarios para la manipulación adecuada del tipo de interés.
  En este caso el género del parámetro requerido se refiere al tipo de objetos que se va a guardar en la estructura que nos ocupa. Las colas pueden guardar objetos básicos (como números o caracteres) u objetos complejos (como registros). Pero de cualquier manera, el tipo de los objetos guardados no influye en el comportamiento de estas colas. Por ello, su esquema de especificación se presenta parametrizado por elementos, para los cuales se ha definido una función de orden, donde los elementos están representados por toda una clase que cumple con las propiedades de la especificación del parámetro **ElementosOrd**.
  En la aplicación correspondiente, el elemento será sustituido por el tipo concreto que se requiera.

- La **especificación** propiamente dicha, que introduce el tipo que se está definiendo el cual, en este caso corresponde a **ColaPr** (cola con prioridades), junto con la definición sintáctica de las operaciones que lo manipulan, y la descripción abstracta de la semántica de las operaciones a través de los **axiomas**, los cuales describen las propiedades de estas operaciones mediante ecuaciones.

A continuación, se profundizará un poco más en las operaciones y los axiomas tanto del parámetro **ElementosOrd** como del tipo de interés definido **ColaPr**.

2.431 Parámetro: ElementosOrd

Inicialmente, en la clausula de importación se mencionan las relaciones que se requieren con otros módulos; en este caso se requiere el tipo **Bool** de **Booleanos**. De la misma manera, se especifica las operaciones inherentes que esta sección de parámetros permite exportar; en el caso que nos ocupa, se permite exportar **todo**.

A continuación, se cita el género, es decir, el tipo parametrizable **Elem**.

En el caso particular de colas con prioridades, el parámetro **elementos** debe proporcionar una relación de orden que lo convertirá en **ElementosOrd**. Para ello se definen dos funciones lógicas (**igual** y **menor que**) que reciben dos valores de tipo elemento y devuelven un valor de tipo booleano, lo cual queda expresado de la siguiente manera:

_=_ :Elem*Elem → Bool;

_<_ :Elem*Elem → Bool;

A su vez, los axiomas correspondientes a los parámetros definen las propiedades de estas relaciones. Por ejemplo, el axioma que establece: **(e1=e2) = (e2=e1)** expresa que la relación de igualdad es simétrica, en tanto que el axioma que declara: **(e1<e2) ⇒ (e2<e1)=falso** establece la no simetría de la relación "menor que".

2.432 Tipo de interés: ColaPr

Cláusulas de composición y extensión

Al igual que en el caso del parámetro, inicialmente se citan las cáusulas de importación y exportación necesarias para componer y extender los módulos. En la sección de la especificación, se menciona como importación necesaria todo lo relativo al parámetro **ElementosOrd**. Se permite la exportación de todas las operaciones relativas a la especificación.

Género

Se establece el género, es decir el tipo que se está definiendo que, en este caso, es **ColaPr**.

Operaciones

Existen tres tipos de operaciones aplicables a un tipo de datos abstracto: operaciones **constructoras** o **generadoras** del tipo de interés (por medio de las cuales se obtienen nuevas estructuras a partir de una elemental), operaciones de **extensión** (que no son necesarias para construir nuevas estructuras) y operaciones **observadoras** (que permiten conocer situaciones específicas de las diferentes estructuras, sin afectar su naturaleza intrínseca).

Como **operaciones constructoras** del tipo de interés se incluyen **vacia** y **mete**.

La cola con prioridades más simple es la vacía, la cual se representa mediante una operación constante (sin dominio) que siempre genera el objeto **cola con prioridades**:

vacia: → ColaPr;

Dada una cola con prioridades **c**, se puede construir otra cola con prioridades el insertando un elemento a la cola **c**. La acción de meter se puede considerar como una operación que, dada una cola **c** y un elemento **e**, devuelve como valor una cola nueva:

mete:Elemento*ColaPr → ColaPr;

Esto se lee así: "Operación **mete** recibe dos argumentos, uno de tipo Elemento y otro de tipo ColaPr; devuelve un valor que pertenece al tipo ColaPr."

Como **operaciones observadoras** se tienen **esvacia** y **maximo**. La primera permite distinguir la cola vacía de una que no lo está y la segunda permite el acceso al elemento con máxima prioridad, el cual se encontrará siempre al principio de la cola:

esvacia:ColaPr → Bool;

maximo:ColaPr → Elem;

Como **operación de extensión**, que genera valores del tipo pero que no es indispensable para construirlos, está **sacamax**, función que se encarga de sacar el valor de mayor prioridad:

sacamax: ColaPr → ColaPr;

Axiomas

Los axiomas definen las propiedades que deben cumplir estas operaciones:

El resultado de aplicar la función **esvacia** a una cola vacía debe ser cierto. El resultado de aplicar la función **esvacia** a una cola a la que se le acaba de meter un elemento debe ser, por supuesto, falso. Esto está expresado en los axiomas 5 y 6:

esvacia(vacia)=cierto
esvacia(mete(e,c))=falso

Se considera que no tiene ningún sentido aplicar las funciones de **maximo** y **sacamax** a la cola vacía y dado que se supone que existe un valor específico de error que pertenece a todos los tipos y que expresa la aplicación indebida de operaciones válidas, se define esta situación en los axiomas 1 y 3 de colas con prioridades:

sacamax(vacia)=error
maximo(vacia)=error

Las especificaciones adoptadas suponen que todas las operaciones son estrictas; de esta manera, si alguno de los argumentos de una operación tiene el valor de error, esta operación devolverá el valor de error. Por ejemplo:

esvacia(sacamax(vacia))=esvacia(error)=error.

significa que tratar de determinar si está vacía una cola con prioridades vacía, a la cual se le trató de sacar el elemento con valor de prioridad máximo, dará como resultado un error. Esto se debe a que, al resolver la función anidada más interna (**sacamax(vacia)**), se obtuvo como resultado un error; aplicar la siguiente función con un argumento de error (**esvacia(error)**) dará por resultado nuevamente error.

El axioma 4 precisa que el elemento de máxima prioridad que se obtiene de una cola **c** a la que previamente se insertó un elemento **e** (expresado como **maximo(mete(e,c)**) tiene 2 posibilidades:

- Si la cola original **c** estaba vacía o el elemento de máxima prioridad de la cola original era menor al elemento **e** que se insertó, entonces devolverá como resultado el valor del último elemento que se insertó:

  SI esvacia(c) o maximo(c)< e ENT e

- De otra manera (la cola original **c** no estaba vacia o el valor de máxima prioridad de la cola original no era menor al del elemento **e** que se insertó), el valor que devolverá la función será el resultado de aplicar la función **maximo** a la cola original:

  SI_NO maximo(c)

A su vez, el axioma 2 aclara que la cola con prioridades que se obtiene al sacar el elemento con prioridad máxima de una cola **c** a la que previamente se le había insertado un elemento **e** expresado como **sacamax(mete(e,c)**, tiene varias posibilidades:

- Si la cola **c** estaba vacia entonces devuelve una cola vacía (puesto que sacó como elemento máximo el único elemento introducido):

  SI esvacia(c) ENT vacia

- Si **c** no estaba vacía se presentan, a su vez, dos posibilidades:

  ◊ Si el elemento máximo de la cola original **c** era menor al elemento **e**, entonces devuelve la cola original **c** (dado que el elemento **e** introducido fue sacado después como elemento máximo):

  SI_NO SI maximo(c)< e ENT c

◊ En caso contrario (el máximo de la cola c es mayor que el elemento e), devuelve la cola resultante de meter el elemento e en la cola obtenida después de sacar el elemento máxima prioridad de la cola original:

SI_NO mete(e,sacamax(c))

2.44 Corolario

A través de la revisión de los detalles comentados del esquema de especificación algebraica del tipo de datos abstracto **cola con prioridades**, se puede apreciar que se logra expresar:

- La descripción sintáctica del tipo y sus operaciones mediante la **signatura**, la cual consiste en la definición de los **nombres** de los tipos involucrados (en este caso, el tipo parametrizable **Elem** y el tipo que se está definiendo **ColaPr**, junto con la definición sintáctica de las **operaciones** que manipulan a los objetos del tipo de interés.

- La descripción abstracta de la semántica a través de los axiomas.

- La adecuada modularización, porque cada tipo de datos abstracto constituye una unidad autosuficiente, que realiza un solo tipo de operaciones -en este caso, las referentes a las colas con prioridades- y que depende lo menos posible de otros módulos. Con ello se espera que el módulo tenga una cohesión fuerte y un acoplamiento débil.

- La encapsulación, ya que reúne en una sola ubicación una estructura de datos con las únicas operaciones legales que las pueden afectar, permitiendo que se conozca qué es una cola con prioridades pero sin establecer cómo debe ser implementada en algún lenguaje particular de programación.

- La composición y extensión de cápsulas por medio de las cláusulas de importación y exportación que permiten combinar las especificaciones de otros módulos (como en el caso de la importación del tipo **Bool** de **Booleanos**), o ampliar las características básicas de una cápsula ya definida (como en el caso del tipo **Elem** al cual se definen dos operaciones específicas de comparación para conformar el parámetro **ElementosOrd**) sin que sea necesario repetir todo el código correspondiente.

- La **parametrización** de cápsulas que, en este caso, se refiere al tipo de objetos que se va a guardar en la estructura que nos ocupa (esto es, **ElementosOrd**).

Según se recordará, todos estos aspectos fueron mencionados como el requisitos de expresión que debería tener el lenguaje adecuado para definir **tipos de datos abstractos**.[11]

[11] Punto 2.32 de este mismo capítulo.

3. IMPLEMENTACIÓN DE ESPECIFICACIONES ALGEBRAICAS

Las especificaciones algebraicas de tipos de datos abstractos sirven como una descripción de lo que el usuario puede utilizar en sus implementaciones. El programador toma esta especificación como el punto de partida para elaborar una implementación.

Para lograr la implementación, primero se debe seleccionar un lenguaje de programación; posteriormente, se debe seleccionar la representación de los datos del tipo abstracto mediante estructuras de datos permitidas por el lenguaje, para después codificar las operaciones en forma tal que se satisfaga la definición abstracta.

En este capítulo se tratarán las características deseables que un lenguaje de programación debe ofrecer con el fin de ser considerado como una posibilidad para implementar las especificaciones construidas. Bajo esa caracterización se evaluará el lenguaje Ada, que es el que se ha seleccionado para llevar a cabo dicha implementación. Asimismo, se mencionarán las estructuras disponibles en este lenguaje de programación.

3.1 CUALIDADES DEL LENGUAJE DE PROGRAMACIÓN IDEAL PARA LA IMPLEMENTACIÓN

Los lenguajes de programación son los vehículos adecuados que permiten implementar las especificaciones producidas durante la etapa de diseño, traduciéndolas a formas que puedan ser comprendidas por la computadora.

En términos generales, la selección del lenguaje de programación se hace tomando en cuenta dos tipos de criterios:

- criterios de orden teórico -relacionados básicamente con los principios de la **Ingeniería de Software**- que consideran características del lenguaje mismo, y
- criterios de orden práctico, que se derivan del ambiente en que se va a trabajar.

Todas estas características afectan la facilidad con que se implementará el diseño y el esfuerzo requerido para llevar a cabo la prueba y el mantenimiento del *software* lo cual, finalmente, repercutirá en la calidad del producto obtenido.

En cuanto a los criterios específicos no existe consenso entre los estudiosos[1]; sin embargo, siempre está presente, de alguna manera, la consecución de las metas de la **Ingeniería de Software** (esto es, que el *software* resultante sea comprensible, confiable, eficiente, mantenible, modificable y transportable) a través de la aplicación de los principios en que se basa dicha disciplina (abstracción, ocultamiento de información, modularidad, localización, uniformidad, completez).

Entre los criterios de orden teórico se mencionan aspectos tales como las facilidades que debe ofrecer un lenguaje de programación para lograr los siguientes objetivos: adecuación a la aplicación, capacidades de abstracción, compilación por separado, estructuras de control, facilidades para la entrada/salida, interfaz con el *hardware*, interfaz hacia otros lenguajes, manejo de excepciones, mecanismos de concurrencia, posibilidades de ejecución parcial del código, portabilidad, posibilidades de recibir mantenimiento, entre otros.

Entre los criterios de orden práctico se deben tomar en consideración diversas características como por ejemplo: disponibilidad de compiladores, disponibilidad de componentes de biblioteca, disponibilidad de herramientas, exigencias del cliente, estandarización del lenguaje, lenguajes de implementación de proyectos anteriores, nivel de experiencia del equipo de programación, etcétera.

En este caso, se desea lograr una implementación de **tipos de datos abstractos** que sea modular y reusable, por lo que se requerirá que el lenguaje seleccionado ofrezca facilidades que permitan la creación y faciliten el reuso de componentes; esto es, que ofrezca estructuras que permitan el manejo de los siguientes conceptos:

3.11 Modularidad

Un sistema complejo difícilmente puede ser desarrollado si se le trata como un gran bloque. Para facilitar su análisis, diseño, desarrollo y mantenimiento es necesario descomponerlo en unidades lógicas relacionadas a las que se acostumbra llamar "**componentes**" o "**módulos**"[2] .

---

[1] [PRESS89], [BOOCH87], [JONES90], [WIENE84], [MEYER88]

[2] Ver punto 1.21 Sistemas basados en composición.

Un módulo es una agrupación de elementos lógicamente relacionados. La descomposición de un sistema en módulos debe efectuarse bajo una metodología que facilite la producción de componentes que puedan ser fácilmente combinados con otros, bajo reglas específicas, para producir el sistema deseado.[3]

El lenguaje de programación debe proporcionar mecanismos para que los módulos cuenten con interfaces eficientes que permitan la comunicación entre componentes compatibles. Esto facilitará la descomposición-composición de sistemas más complejos, a partir de partes reusables, así como posibles modificaciones o extensiones al mismo.

Lo anterior implica contar con la facilidad de compilación por separado, es decir, que las unidades de un programa se puedan compilar separadamente para, posteriormente ser integradas para formar un programa completo. La facilidad de compilación independiente permite la ingeniería de *software* a gran escala y simplifica su mantenimiento al requerir que sólo las unidades modificadas sean recompiladas.

La creación de sistemas basados en composición aprovecha esta característica para construir un sistema complejo sobre módulos o componentes ya existentes y probados, los cuales pudieron ser programados por personas o equipos de programadores diferentes. Así, en vez de definir todas las características de un componente, basta con mencionar su inclusión para que quede incorporado de una manera adecuada.

La modularidad no sólo da soporte al reuso de componentes en todo un sistema sino que además facilita la extensión del mismo, minimizando la cantidad de código que debe agregarse o cambiarse en él.

Existen estructuras similares en lenguajes como Modula-2, mediante los llamados "**módulos**" que se combinan mediante importaciones y en Ada que cuenta con los "**paquetes**" y los "**subprogramas**" que se pueden asociar mediante las cláusulas de contexto "**with**" y "**use**".

3.12 Encapsulación

Es el mecanismo que facilita la definición y uso de un **tipo de datos abstracto** como una unidad lógica, en la que un tipo de datos tiene asociado un conjunto básico de operaciones que se aplican a los objetos de ese tipo. Esto permite concentrar la atención del usuario en las propiedades de esas entidades, su estructura visible y el conjunto de operaciones que se pueden realizar con ellas, sin que tenga que preocuparse del detalle relativo a cómo los lleva a cabo el programa correspondiente.[4]

La encapsulación generalmente se usa para proteger los datos privados de un objeto del posible acceso del exterior. En lugar de organizar programas mediante procedimientos que comparten datos globales, los datos son empacados con los procedimientos que accesan esos datos. La intención es separar al usuario de un objeto del implementador del mismo.

Al permitir sólo ciertas operaciones y prevenir cualquier operación que pudiera violar la lógica del nivel, evitando el uso de datos compartidos (datos globales) que pudieran ser afectados incorrectamente por diversas unidades del programa, se asegura la confiabilidad de los sistemas de *software*[5] a través de la reducción de interdependencias entre sus componentes.

---

[3] [QUINT93a]
[4] Con dice que, desde esta perspectiva, un "objeto" (dentro del paradigma orientado a objetos) es una estructura de datos encapsulada. [COX85] cit en [PRESS89]
[5] [BOOCH87],[WIENE84]

Un componente encapsulado está constituido por dos partes básicas: la especificación y el cuerpo. La especificación o interfaz es la parte exterior, visible, disponible para el usuario. Permite exportar el nombre del componente y las únicas operaciones definidas para él. Con sólo ver esta vista exterior, un módulo puede interactuar con otro sin conocer la representación o implementación del otro. El cuerpo o parte privada del componente contiene la implementación del comportamiento de ese módulo, la cual se mantiene oculta de la vista del exterior.

Se pueden encontrar estructuras que permiten la encapsulación en las **clases** del lenguaje Simula, en los **objetos** de Smalltalk, en los **clusters** de CLU, en las **estructuras** y los **functores** de ML, en los **módulos** de Modula-2 y de Mesa, así como en los **paquetes** de Ada.

3.13 Ocultamiento de la Información

Se trata de una idea propuesta originalmente por Parnas que sugiere descomponer a los sistemas en base al principio de ocultar las decisiones tomadas en el desarrollo de un diseño.[6]

El propósito del ocultamiento de la información es hacer inaccesibles ciertos detalles que no deben afectar otras partes del sistema, detalles relacionados con la representación interna de los tipos de datos de un componente.

Presente en los objetos de Smalltalk, en los **tipos de datos abstractos** y las **signaturas** de ML y en los **tipos opacos** de Modula-2, el ocultamiento de la representación se hace en Ada en forma explícita mediante el uso de objetos que se especifican como **tipos privados** o **tipos privados limitados**.

3.14 Genericidad

Genericidad es una técnica para definir elementos que tienen más de una interpretación, dependiendo de los parámetros que reciban, los cuales representan tipos.

Se trata de una característica que promueve la reusabilidad, la extensibilidad y la compatibilidad de los componentes de *software* implementados en lenguajes tipificados estáticamente y que aplica una forma de **polimorfismo**, concepto que puede explicarse como la posibilidad de definir unidades de programas que pueden ser aplicados a diferentes tipos de datos.

Una forma sencilla de polimorfismo es la **sobrecarga**, que es la característica que permite asociar más de un significado a un mismo nombre y cuyas ambigüedades se resuelven mediante el examen del contexto de cada ocurrencia.

La genericidad representa la posibilidad de definir módulos parametrizados. Estos módulos, llamados módulos genéricos, no se pueden usar en forma directa ya que en realidad son patrones de módulos cuyos parámetros, llamados parámetros genéricos formales serán sustituidos por tipos específicos. Los módulos reales, llamados **instancias** de los módulos genéricos, se generan en el momento en el que se proporcionan tipos actuales (parámetros genéricos actuales) para cada uno de los parámetros genéricos formales.

El uso de unidades genéricas proporciona un medio de parametrizar definiciones, de tal manera que puede darse una sola definición para una clase de objetos.

---

[6] [BOOCH86]

Ada ofrece esta característica a través de sus **subprogramas genéricos** y sus **paquetes genéricos**, al igual que otros lenguajes de programación como CLU y LPG, característica que corresponde al enfoque de parametrización explícita. Algunos lenguajes de especificación formal como Z, Clear, OBJ y LESPAL, también la tienen.

Resulta interesante mencionar que, para ML y otros lenguajes funcionales, se desarrolló una variante de este enfoque, llamada genericidad implícita, la cual permite que el programador omita las declaraciones de tipo cuando no son conceptualmente necesarias, haciendo que el compilador sea el que verifique que todos los usos de un identificador son consistentes.[7]

3.15 Manejo de Excepciones

Un sistema difícilmente es perfecto, hay tolerancias; pero al menos se espera que un sistema al fallar se degrade sin causar efectos laterales peligrosos.

Para ser confiable un sistema debe prever tanto fallas de diseño y construcción, como la forma de recuperación de fallas en operación.

Al menos para aplicaciones en tiempo real, el lenguaje de programación que se utilice en la implementación deberá proveer facilidades para definir las consecuencias de error, disfuncionamientos o condiciones de datos inesperadas; deberá dar también facilidades para especificar el comportamiento de recuperación del sistema para casos no previstos.

El manejo de excepciones, por sí mismo, no constituye una técnica para resolver errores, pero al menos permite que el programador controle la secuencia de acciones que se sucederán una vez que se presente el caso.

Meyer[8] distingue 3 casos en los que el mecanismo de excepciones juega un papel indispensable:

- Casos anormales que llevan a acciones drásticas del *hardware* o del sistema operativo, como en los casos de sobreflujo numérico o falta de memoria.
- Casos anormales que llevan a la terminación inmediata de un programa para evitar consecuencias potencialmente lamentables.
- Tolerancia a fallas de *software* : protección contra posibles errores en algún componente del propio sistema.

Ada es uno de los pocos lenguajes de programación que proporciona un manejador de excepciones como una técnica para tratar las situaciones de error. Separa la detección de errores del manejo de los mismos.

3.16 Portabilidad

La disponibilidad de compiladores del lenguaje accesibles que además respondan a una estandarización es una garantía respecto a su portabilidad y, en consecuencia, las posibilidades de aceptación, utilización y permanencia de los componentes producidos. Esta garantía repercutirá en un menor costo de producción y mantenimiento de los programas correspondientes.

---

[7] [MEYER88]
[8] [MEYER88]

3.2 CARACTERÍSTICAS PRESENTES EN ADA

Ada es un lenguaje moderno que, en el momento de su aparición, en los años ochenta, fue resultado de un proyecto de Ingeniería de *Software* promovido por el Departamento de Defensa de Estados Unidos -el mayor usuario de computadoras en el mundo- para responder al conjunto de requerimientos bastante bien definidos, conocido como Steelman, el cual enfatiza las características de un lenguaje que soporte:

- Componentes estructurados.
- Tipificación estricta.
- Especificación de precisión relativa y absoluta.
- Ocultamiento de la información y abstracción de datos.
- Procesamiento concurrente.
- Manejo de excepciones.
- Definición genérica.
- Facilidades dependientes de la máquina.

De esta manera, el lenguaje resultante -Ada- incorporó prácticamente todos los adelantos y atributos de los lenguajes considerados como útiles e importantes para la ingeniería de los sistemas de *software* a gran escala.

Se citará a los autores: Wiener[9], Habermann[10] y Sammet[11] quienes describen al lenguaje Ada en términos muy elogiosos y que, en cierta forma, resumen la opinión de diversos autores[12]:

> "Ada directamente incorpora, estimula y refuerza los principios y las metodologías de la ingeniería de software modernos. Ada no es sólo un lenguaje sino también un ambiente de programación y una manera de pensar. Ada conjunta la mejor tecnología de programación en una forma coherente para satisfacer las necesidades de los programadores técnicos. Se espera que Ada reduzca los costos del ciclo de vida del software, que promueva la inversión en tecnología de soporte, que mejore la adaptabilidad del personal de software , que promueva el desarrollo de software reusable y confiable y que promueva a la Ingeniería de Software como una disciplina."
>
> "Ada es un lenguaje rico. Tiene múltiples características que apoyan al programador en el diseño y mantenimiento de grandes sistemas. Sin embargo, esto no implica que la compilación de los programas de Ada dé como resultado un código de máquina muy complicado. El énfasis que pone el lenguaje se da en las verificaciones a tiempo de compilación y en las declaraciones para la elaboración de los formatos de los objetos. Gran parte de la expresividad que tiene a nivel de diseño se simplifica a la hora de la compilación sin dejar rastro en el código objeto."
>
> "El carácter especial de Ada se basa en el soporte que da al concepto de componentes de software, su excelente mezcla de útiles características modernas y su soporte a la producción de sistemas de software muy grandes."

---

[9] [WIENE84]
[10] [HABER83]
[11] [SOMME87]
[12] [BOOCH87], [DUBIN87], [GOLD685], [OCH893], [SAMME86], [WICHM84]

Se hará una breve reseña de las estructuras de Ada, referentes a la descripción de las características mencionadas en el punto anterior, las cuales permiten la implementación de tipos de datos abstractos en las condiciones señaladas

3.21 Ambiente de Programación

La estandarización de un lenguaje proporciona un mecanismo formal para el control de la evolución de la definición del mismo.

Ada es un lenguaje estandarizado que cuenta con acreditaciones ante ANSI (American National Standards Institute), ISO (International Standards Organization) y MIL-STD (Military Standards through the United States Department of Defense). El estándar original, identificado como ANSI/MIL-STD-1815A-1983, surgió como resultado de la revisión minuciosa del manual de referencia que se dió a conocer en 1980.[13]

Se ha procurado dotar al lenguaje con mecanismos extensivos que faciliten su portabilidad.

Tomando en cuenta que el grado en el que un ambiente de programación completo y coordinado esté disponible repercutirá en el desarrollo de *software* aun más que cualquier lenguaje de programación por si solo, el Departamento de Defensa de Estados Unidos patrocinó el desarrollo de un conjunto de requerimientos que definiesen una colección de herramientas de *software* para dar soporte a las aplicaciones de Ada, a las cuales se les describió en el documento STONEMAN con el nombre de APSE (Ada Programming Support Environment).[14]

El APSE se divide en tres niveles:

- KAPSE.- (Kernel Ada Programming Support Environment).
- MAPSE.- (Minimal Ada Programming Support Environment).
- APSE mismo.- que da al usuario la vista de todas las herramientas.

Aunque no se describe con precisión, el MAPSE debe contener herramientas tales como: editor de textos, embellecedor de impresión, compilador, ligador, analizador estático de control de flujos, herramientas de análisis dinámicos, rutinas de interfaz de terminales, administrador de archivos, intérprete de comandos, administrador de configuraciones.

La intención de estas características ha sido facilitar la labor del programador, favorecer su productividad y respaldar la portabilidad de los sistemas construidos.

El gobierno de Estados Unidos cuenta con programas para el soporte continuado a Ada, entre ellos el programa AVA (Ada Validation Organization) que funciona como una central que aprueba los compiladores Ada.[15] Por otra parte, se creó el programa STARS (Software Technology for Adaptable Reliable Systems) que, utilizando Ada como piedra angular, dirige la atención nacional hacia los aspectos críticos de todas las fases de desarrollo de *software*.

---

[13] Actualmente, el proyecto Ada-9X está realizando un proceso de revisión que incorporará mejoras para dotarlo de características acordes con el estado del avance de la computación. El proceso ha sido largo y minucioso porque se requiere mantener compatibilidad con el estándar vigente.

[14] [BOOCH87]

[15] Para que un compilador pueda ser usado en proyectos gubernamentales debe aprobar más de 2000 pruebas propuestas por la ACVC (Ada Compiler Validation Capability) y recibir el certificado respectivo. Para ayudar al desarrollador de compiladores existe una IO (Implementors Guide).

3.22 Unidades de Compilación

Generalmente, los sistemas en Ada se descomponen en unidades que se compilan por separado sin que se viole la verificación integral que es fundamental en el lenguaje.

Las unidades de compilación pueden estar constituidas por:

- Una o más declaraciones genéricas,
- una o más instanciaciones genéricas,
- una o más declaraciones de subprograma,
- el cuerpo de un subprograma,
- la declaración de un paquete,
- el cuerpo de un paquete,
- una o más subunidades.

Como puede observarse, los subprogramas y los paquetes se pueden compilar por separado; igualmente se puede hacer con el cuerpo de una unidad de programa.

Las reglas de compilación de Ada requieren que una unidad de especificación sea compilada antes de que sea referenciada. Consecuentemente, si se cambia tal especificación, todas las demás unidades de compilación que hagan referencia a ella deben ser recompiladas.

Esto no es necesario cuando el cambio efectuado ocurre en la implementación. El cambio sólo afectará la compilación del cuerpo de la unidad.

Las unidades de compilación de un programa forman parte de la biblioteca de un programa. La biblioteca es el depósito de toda la información existente acerca de las unidades previamente compiladas de Ada que permite verificar la consistencia entre las diversas unidades que integren el programa completo. Formalmente se llama a cada unidad de compilación **unidad de biblioteca** o **unidad secundaria**, que sería el caso de los cuerpos de paquetes y de subprogramas y el caso de las subunidades.

La unidad principal de cualquier programa debe ser un subprograma, el cual por definición será una unidad de la biblioteca del programa.

A diferencia de otros lenguajes, Ada requiere que los compiladores apliquen las reglas de tipos en todas las unidades de compilación. De esta manera se mantiene la protección que ofrece la tipificación estricta, aun en los casos en que los programas sean compilados mediante sesiones múltiples. Aunque esto incrementa la complejidad de los compiladores, resulta benéfico en la construcción de grandes sistemas.

3.23 Asociación de Unidades de Compilación

El alcance de una entidad -parámetro o tipo presentado mediante una declaración- es la parte del código en la cual tiene efecto su declaración. La visibilidad de una entidad define la parte del código en el cual puede ser visto su nombre. Se dice que una unidad es visible directamente si se le puede designar por su nombre simple.

Ada tiene reglas precisas, relacionadas con los conceptos de alcance y visibilidad, que facilitan y apoyan la composición de unidades; las principales son:

- Los elementos declarados dentro de una unidad de programa son locales a esa unidad.
- La referencia a un elemento no declarado dentro de la unidad de programa corresponde al elemento declarado en la unidad más cercana de nivel superior.
- Una vez que un paquete es declarado, su parte es visible a otras unidades de programas, mediante el uso de la cláusula "*use*" o el uso de la notación de punto *(dot notation)*.
- Una unidad de compilación sólo puede ser compilada después de que todas las unidades visibles en la unidad de compilación han sido compiladas.

Para que pueda lograr visibilidad de cualquier otra unidad de biblioteca compilada con anterioridad, la unidad en cuestión deberá aplicar la cláusula de contexto "*with*" y usar una notación de punto con el componente seleccionado o una cláusula "*use*". Esto permite hacer visibles en forma selectiva sólo las unidades que se necesitan usar.

3.24 Sistema de Tipos

Los tipos nos permiten desarrollar unidades conceptuales para modelar objetos del mundo real con precisión y claridad. Subyacente al uso de tipos está la garantía de que las propiedades de los tipos declaradas por el programador no serán violadas durante la ejecución del programa y cualquier violación en los posibles valores de un tipo será reportada.

Ada es un lenguaje con un sistema de tipos estricto. Esto significa que los objetos de un tipo dado sólo pueden tomar valores propios del tipo y que a esos valores sólo se les pueden aplicar las operaciones definidas para ese tipo.

El énfasis que Ada pone en su sistema estricto de tipos responde a diversas necesidades:

- La necesidad de describir objetos con una factorización de propiedades, lo cual facilita en gran medida su proceso de mantenimiento.
- La necesidad de describir en forma explícita las propiedades del objeto, con lo cual el código es más legible.
- La necesidad de garantizar que las propiedades de los objetos no serán violadas, lo que proporciona un margen de seguridad que se refleja en que el código resultante sea confiable.
- La necesidad de ocultar detalles de la implementación, lo cual reduce la complejidad de los programas.

Los sistemas de tipos estrictos permiten detectar más errores durante el proceso de compilación, lo que dará como resultado que el implementador pueda tener más confianza en que el programa se ejecutará correctamente. Para aquellos casos que pudieran resultar con violación a la abstracción definida, Ada cuenta con un mecanismo de detección, mediante excepciones, para evidenciarlos.

Las facilidades para tipos con las que cuenta Ada son muy variadas, tal vez más variadas y poderosas que en cualquier otro lenguaje de programación. En este punto sólo se hará referencia a características que de alguna forma se han asociado con la posibilidad de implementación de los tipos de datos abstractos.

- **Registros con estructura parametrizada**

Ada acepta caracterizar la estructura de un registro con uno o más parámetros llamados discriminantes. Los objetos de este tipo se pueden declarar usando una restricción que discrimina (***discriminant constraint***) la cual constituye una condición que excluye a los registros que no la cumplen.

- **Registros con estructuras alternativas**

En ocasiones, cuando se requiere de una información adicional en los casos en que el componente de un registro adquiere cierto valor, Ada permite su manejo a través de estructuras alternativas que deberán tener un discriminante y una parte variante.

- **Tipos con estructura dinámica**

Cuando se quieren manejar en forma dinámica colecciones de objetos variables, asociados como los elementos de una lista ligada o los integrantes de un árbol familiar, se llega a necesitar crear a los objetos durante la ejecución del programa y, a la vez, expresar dinámicamente su relación con otros objetos. Estos casos se pueden describir con objetos tipo acceso -*access type*- los cuales dan acceso a objetos creados dinámicamente.

- **Subtipos**

Se puede utilizar un subtipo para factorizar y nombrar las limitaciones que se desean definir para un tipo determinado. Sin embargo, un subtipo no introduce un nuevo tipo.

Es importante hacer notar que el tipo de un objeto es estático, lo cual implica que sus características son fijas durante la compilación. En cambio, las características de un subtipo sólo se conocen hasta que el objeto es creado. Al proceso de definición de características no estáticas se le conoce como **elaboración**.

Se obtiene un subtipo nombrando al tipo precedido por la indicación *is*, seguido por el rango de valores permitidos, por ejemplo:

```
subtype Letra is Character range 'A'.. 'Z';

subtype Letra_Hexa is Letra range 'A'.. 'F';
```

- **Tipos derivados**

Cuando se desea expresar la existencia de diversas clases distintas de valores con propiedades y operaciones similares, que no deban ser mezclados sino tratados como conceptualmente diferentes, se puede hacer uso de los tipos derivados.

Generalmente, un tipo llamado B se puede derivar de otro tipo existente llamado A mediante una declaración como la siguiente:

```
type B is new A;
```

Tanto los subtipos, mencionados en el punto anterior, como los tipos derivados permiten la factorización de las propiedades de otros tipos. Sin embargo, los tipos derivados dan origen a nuevos tipos, los subtipos no.

- **Tipos privados**

Dentro de los paquetes, es posible declarar un tipo con la característica de privado, como una salvaguarda para asegurar la integridad de la información, lo que evitará el acceso libre a los componentes de un elemento, acceso que podría ocasionar cambios arbitrarios indeseables.

La únicas operaciones definidas del lenguaje que se pueden efectuar con ellos son la asignación y la comparación.

Si se desea que tampoco estas dos operaciones sean permitidas, se les puede declarar como de tipo privado limitado. El uso de esta forma proporciona control absoluto sobre las operaciones de un tipo.

Los paquetes que introducen tipos privados desempeñan un doble papel. Por una parte evitan que un usuario opere sobre datos definidos en el paquete. Por otra, implementan el concepto de ocultamiento de la información para un tipo de datos abstracto, ya que permite ocultar su representación.

3.25 Paquetes

Los programas grandes se componente generalmente de módulos. En el lenguaje Ada estos módulos se llaman paquetes, los cuales cumplen un doble cometido: permiten particionar un programa grande en piezas más manejables y proporcionar un magnífico vehículo para crear componentes de *software* de propósito general, listos para usarse.

El paquete tal vez sea el recurso más importante de Ada ya que constituye el bloque de construcción básico para un sistema en Ada. Es una estructura que permite agrupar lógicamente elementos de programa que pueden ser usados para expresar lo siguiente:

- Un conjunto de tipos y objetos;
- un conjunto de subprogramas;
- tipos de datos abstractos;
- una máquina de estados abstractos.

Ada hace una clara distinción entre la apariencia externa de un paquete y su desempeño interno. La parte externa, llamada especificación, describe su interfaz la cual incluye: los tipos, las variables, las excepciones del paquete, así como los nombres, tipos de parámetros y tipos de los resultados de los subprogramas presentes en el paquete. La parte interna, llamada cuerpo del paquete, describe su implementación, la cual puede incluir: los cuerpos de los subprogramas, subprogramas internos -inaccesibles desde el exterior pero usados para implementar a los otros subprogramas-, variables, excepciones y tipos a los cuales se hace referencia sólo dentro del cuerpo del paquete.

Para el usuario del paquete sólo es importante la parte de la interfaz.

El paquete puede introducir un tipo privado, lo cual significa que la estructura del mismo no será parte de la interfaz. Los valores de un tipo privado pueden ser manipulados de acuerdo a su estructura sólo desde dentro del paquete mismo. Esto tiene la ventaja de que se evita una posible corrupción indeseable y se limita la cantidad de información que importa conocer externamente, con lo que se simplifican las interconexiones entre los módulos de un programa extenso.

La construcción de paquetes puede contribuir notablemente a lograr una ingeniería de programas efectivos y de fácil mantenimiento.

3.26 Unidades Genéricas

Ada da la facilidad de usar unidades genéricas que son, básicamente, patrones o esquemas para paquetes o subprogramas con algunas partes que se dejaron sin especificar.

Una unidad genérica no puede llamarse directamente. Es un modelo computacional que debe instanciarse en el momento de la compilación. El proceso de **instanciación** implica la sustitución de la información faltante correspondiente a los parámetros genéricos.

La instancia resultante podrá ser usada como un paquete o subprograma común. Un patrón único podrá dar origen a diversas instancias en las cuales se especifique información diferente.

Las características genéricas de Ada constituyen un mecanismo muy poderoso para extender el rango de aplicación del lenguaje ya que permiten la construcción de componentes de *software* de uso general que pueden instanciarse para resolver una variedad de problemas similares aunque no idénticos.

3.27 Sobrecarga

En Ada, los nombres de los operadores aritméticos y de los subprogramas pueden sobrecargarse, en tanto el uso del nombre no sea ambiguo. Esto significa que, dos nombres iguales darán origen a asociaciones diferentes por parte del compilador que se determinarán, dependiendo del contexto, por medio del número, modo y tipo de los parámetros incluidos en la solicitud.

La sobrecarga, que constituye una forma simple de polimorfismo, ayuda tanto en el manejo de la abstracción como en el espacio de memoria que requiere el procesamiento de un nombre. La abstracción se logra porque el mismo nombre de subprograma puede utilizarse para operaciones conceptualmente equivalentes que pueden actuar sobre tipos de datos diferentes. El manejo de nombres se simplifica debido a que se requiere un número menor de nombres, con lo cual pueden simplificarse las convenciones de nominación.

La sobrecarga de operadores facilita la producción de programas legibles y apoya la extensión del lenguaje, al permitir definir nuevos significados para los operadores ya existentes y, además, contribuye considerablemente a la creación de bibliotecas de programas.

3.28 Manejo de Excepciones

Ada fue diseñado para codificar programas altamente confiables, capaces de responder de una manera sensible a situaciones inesperadas. A esas situaciones inesperadas en Ada se les llama **excepciones** y las respuestas correspondientes se realizan a través de **manejadores de excepciones** específicos.

De acuerdo con la terminología de Ada, una excepción designa al evento que causa la suspensión de la ejecución normal de un programa. El evento puede ser un error o una situación excepcional que requiere atención especial. En el mejor de los casos, se desearía que el programa fuera capaz de responder a la excepción; en el peor caso, en el cual la recuperación total está fuera del control del programa, sería deseable un degradamiento "*controlado*", que permitiera continuar el proceso aún con capacidad aminorada.

Levantar una excepción, "*raise*", llama la atención hacia la condición. El manejo de la misma, "***exception handler***" es la respuesta que se genera.

Ada cuenta con una serie de condiciones de excepción predefinidas, declaradas en un paquete llamado **Standard**, que son activadas por el sistema a tiempo de ejecución bajo ciertas condiciones, pero además permite que el usuario las pueda activar explícitamente o que defina sus propias excepciones.

Las excepciones pueden ser manejadas dentro del mismo bloque. Si una excepción no contiene a su manejador, ésta será propagada a la estructura de bloque siguiente.

El manejo de excepciones, que generalmente se proporciona, da varias posibilidades como: abandonar la acción, reintentar llevarla a cabo, usar un método alternativo o reparar la causa que generó la excepción.

4. IMPLEMENTACIÓN EN ADA

El lenguaje de especificación utilizado para definir los tipos de datos abstractos que se desea implementar es fundamentalmente algebraico. Se trata de ***LESPAL***, lenguaje en español, basado principalmente en conceptos de van Horebeeck.

El lenguaje de programación elegido para implementar esas especificaciones algebraicas es **Ada**. La elección se hizo considerando todas las características que presenta, las cuales le permiten apoyar fuertemente los principios de la Ingeniería de Software.

Una vez seleccionado el lenguaje de programación, fue necesario escoger, de entre todas las estructuras y operaciones disponibles en el lenguaje, aquéllas que estuviesen más de acuerdo con el estilo de implementación que se pretendía lograr.

Durante la implementación, se siguieron muchas de las propuestas, sugerencias y recomendaciones señaladas por los diversos autores consultados, pero muy especialmente por Grady Booch y por Rachel Harrison[1]. Sin embargo, en los casos en que hubo diferencias, el criterio que prevaleció fue el de cumplir con los requerimientos señalados por las especificaciones algebraicas adoptadas.

En este capítulo se hará la reseña de aspectos generales de las decisiones de implementación tomadas que culminará con un esquema que presenta los componentes implementados.

[1] [BOOCH87] y [HARR89]

4.1 Versión estandarizada

Como ya se mencionó en el capítulo anterior, Ada es un lenguaje que cuenta con acreditaciones ante *ANSI* (American National Standards Institute), *ISO* (International Standards Organization) y *MIL-STD* (Military Standards through the United States Department of Defense).

El estándar original identificado como *ANSI/MIL-STD-1815A-1983*, al cual se suele hacer referencia como LRM (Language Reference Manual)[2], contiene todos los lineamientos que han regido desde su publicación en 1983 hasta la fecha.

4.2 Compilador

Para el desarrollo, codificación y prueba de los componentes implementados se utilizó el compilador *Meridian AdaVantange* para sistemas MS-DOS o PC-DOS, versión 2.1 liberada en 1988 por Meridian Software Systems, Inc.

4.3 Estructuras seleccionadas

Con la finalidad de facilitar la lectura y comprensión de la lectura del código generado, se procuró ser consistente en el estilo seguido en la implementación de todos los módulos o componentes bajo los siguientes lineamientos:

- La implementación de las clases de tipos de datos abstractos (listas, pilas, colas, etc.) se hizo a través de **paquetes genéricos** que son la estructura idónea para modularizar el diseño, encapsular a un tipo de dato abstracto y ocultar los detalles de su funcionamiento.

- Naturalmente, la implementación resultante sólo es una plantilla o patrón que contiene todas las características importantes del comportamiento del tipo de dato abstracto correspondiente pero que debe ser **instanciada** antes de poder ser usada como objeto real.

- Cada clase exporta solamente una declaración de tipo la cual coincide con el nombre del paquete genérico.

- Los objetos (listas de caracteres, listas de registros, pilas de enteros, pilas de cadenas, etc.) se generan como resultado de la instanciación de un tipo de dato específico. En el ejemplo, las instanciaciones específicas se hacen en el programa de validación del caso particular para **colas con prioridades** (que se presenta dentro de la sección 4.22 de este mismo capítulo).

- El ocultamiento de la información se maneja a través de la declaración de **tipos privados** (especialmente para la importación del tipo elemento) y de **tipos privados limitados** (sobre todo para seleccionar la representación del tipo parametrizable).

- Para el manejo de campos de información que pueden guardar diferentes tipos de datos, según se requiera, se usan **registros con variantes**, como en el caso de las

[2] [LRM83]

listas generales que, al estilo de **Lisp**, deben permitir el ingreso de elementos y de otras listas como integrantes de una lista mayor.

- Para implementar estructuras de datos no acotadas se utilizan apuntadores a nodos, en tanto que para las estructuras acotadas se emplean arreglos.

4.4 Tipos de operaciones

- La visibilidad se define estáticamente a través de la cláusula de contexto **"with"**. A pesar de repercutir en el tamaño del código del programa, se prefiere usar la notación de punto, evitando el empleo de la otra cláusula de contexto **"use"**, para facilitar la lectura del programa.

- La importación de operaciones requeridas por una clase se expresa mediante subprogramas genéricos formales.

- Las operaciones inherentes a cada clase se expresan como subprogramas exportables, incluidos en la especificación o interfaz del paquete.

- Las operaciones necesarias internamente para que el paquete pueda realizar las acciones requeridas por el tipo de datos abstracto se declaran e implementan dentro del cuerpo. Naturalmente no son exportables.

- Con el fin de facilitar la identificación de la operación de la especificación algebraica con la operación correspondiente dentro de la interfaz o especificación del paquete, normalmente se le da un nombre igual o muy similar al de su antecedente.

- Las **operaciones constructoras** o **generadoras** se implementan como procedimientos ya que con esto se facilita su anidamiento y su aplicación en aquellos casos en que se requiere recursión.

- Las **operaciones selectoras** u **observadoras** se implementan como funciones.

4.5 Errores

- Las situaciones que dan origen a errores se expresan en Ada como **excepciones**, aprovechando la estructura disponible en este lenguaje de programación. La propagación de excepciones en Ada coincide con el carácter estricto expresado por los axiomas de la especificación.

- La **declaración** de excepciones se realiza dentro de la interfaz del paquete genérico. El señalamiento de su **activación** (**"raising"**) se incluye dentro del cuerpo del mismo.

- La **implementación** de los **manejadores** de **excepciones** corresponde al usuario de cada paquete genérico, una vez que han creado **instancias** de él. En el ejemplo de **colas con prioridades** que se incluye, el código de los manejadores de excepciones se presenta dentro del cuerpo del programa de validación.

- Siempre que es posible, se aprovechan las excepciones intrinsecamente definidas por el lenguaje, con la convicción de que la eficacia y rapidez de su activación superan cualquier otra activación externa. Sin embargo, se trata de encauzarlas en alguna forma y no simplemente dejar que sean propagadas hasta el nivel más alto, lo cual llevaría a una terminación de la ejecución sin control.

4.6 Iteradores

Los iteradores son operaciones observadoras que permiten visitar cada uno de los elementos componentes del tipo de datos sin afectar su estado actual.

Las especificaciones algebraicas no las contemplan en forma específica porque no constituyen operaciones indispensables para el funcionamiento adecuado del tipo de datos abstracto.

En realidad se pueden visitar todos los elementos de una estructura cualquiera por medio de sus operaciones básicas.

Sin embargo, en algunos casos no es tan sencillo. Por ejemplo, en el caso de una pila no acotada, la operación se lograría aplicando la operación **tope** para leer el último elemento insertado, **saca** para pasarlo a otra pila temporal y tener acceso al elemento siguiente, repitiendo la operación hasta llegar al primer elemento insertado; esto implicaría, a su vez, sacar los elementos de la pila temporal para reintegrarlos a la pila original.

Este procedimiento resulta complicado y la operación práctica requiere de un mecanismo que permita visualizar, fácilmente, todo el detalle del estado interno de la estructura.

Existen dos formas para declarar e implementar iteradores: activa (que se presenta como un conjunto de operaciones primitivas) y pasiva (que se logra a través de una operación única que requiere de otra función específica la cual deberá ser implementada en el programa de aplicación).

En el caso relativo a la iteración, se siguen los siguientes lineamientos:

- Sólo se implementa un **iterador** explícito en los tipos de datos abstractos en los que no es posible lograr el acceso mediante la aplicación de otras de sus operaciones primitivas, sin afectar su estado actual.

- Aunque se codificaron tanto formas activas como pasivas, la forma seleccionada corresponde a la de **iterador pasivo** por considerarse que es la que mayor seguridad ofrece, además de que es la que da como resultado una solución más elegante, compacta y legible.

4.7 Recolección de basura

Las formas acotadas de los componentes no requieren de ninguna previsión para la recolección de basura. Las formas no acotadas pueden requerirla.

Ciertos compiladores cuentan con alguna forma de recolección automática. Tal es el caso del compilador **Meridian AdaVantage** usado, el cual tiene lineamientos aplicables como los que se mencionan a continuación.[3]

- Los objetos locales sólo son direccionables en tanto estén dentro del alcance de un subprograma activo.
- El total de espacio de memoria asignado a objetos direccionados con ***new*** y objetos direccionados internamente, como los arreglos dinámicos y los arreglos con discriminantes, está limitado solamente por la memoria disponible.
- Las listas ligadas y otras estructuras de datos dinámicas que no están sujetas a limitaciones de espacio de pila o de datos globales pueden ser creadas fácilmente mediante direccionadores (new) y destruidas mediante la aplicación del procedimiento genérico **unchecked-deallocation**.
- El esquema de manejo dinámico de memoria, empleado por el sistema a tiempo de ejecución del compilador ***Meridian AdaVantage***, hace uso de las llamadas del sistema operativo ***DOS*** para direccionar y liberar memoria, por lo que depende totalmente de las facilidades proporcionadas por él.

No se consideró necesario crear un módulo específico que se encargara de efectuar la recolección de basura generada por la operación de estructuras de datos no acotadas porque, como puede apreciarse, el lenguaje en sí se ocupa de liberar la memoria asignada a objetos a los que ya no se puede hacer referencia directa.

4.8 Formas implementadas

De una manera muy general, los componentes implementados pueden caracterizarse como correspondientes a las siguientes formas:

- Secuenciales.- Todos los componentes implementados ejecutan acciones en serie. No se consideró ningún paralelismo potencial (el cual se puede expresar en Ada a través de las estructuras llamadas "**tareas**").
- Acotadas y No acotadas.- Las formas acotadas se implementan a través de registros. Para las formas no acotadas se emplea el manejo dinámico de memoria por medio de la creación de nodos señalados por apuntadores.
- Sin iterador explícito y con iterador explícito.- Como ya se mencionó en el punto 4.14, sólo se crearon módulos con iteradores explícitos en aquellos tipos de datos abstractos en los que no se pudo lograr acceso mediante el uso de operaciones primitivas, sin afectar su estado actual.
- Sin recolector de basura.- Ninguno de los componentes incluye la posibilidad de activar explícitamente la acción de recolección de basura.
- Desordenadas o con ordenamiento implícito. Los tipos de datos abstractos implementados no contemplan acciones de ordenamiento salvo en los casos en que

[3] [MERID88] Chapter 9

su definición abstracta así lo especifica, como es el caso de colas con prioridades, de ciertas formas de árboles y de conjuntos y tablas, para los cuales se define una función genérica de ordenamiento.

A continuación se presenta el esquema de componentes que señala las formas específicas que contiene su implementación.

TIPO DE DATO ABSTRACTO	ACOTADO	ITERADOR EXPLÍCITO	ORDENAMIENTO IMPLÍCITO
1. Listas			
2. Listas generales			
3. Pilas		X	
4. Pilas acotadas	X		
5. Colas		X	
6. Colas acotadas	X		
7. Bicolas		X	
8. Colas con prioridades		X	X
9. Árboles binarios			
10. Árboles binarios extendidos			
11. Árboles de búsqueda binaria			X
12. Árboles AVL			X
13. Tablas Hash de diccionarios			
14. Conjuntos		X	X
15. Multiconjuntos			

5. UN CASO PARTICULAR: COLAS CON PRIORIDADES

El caso particular que se va a detallar es el de **colas con prioridades**. Este módulo fue seleccionado porque contiene tanto un iterador explícito como un ordenamiento implícito, lo cual permite ilustrar en forma más extensa cómo se logra su implementación.

La definición, el diagrama de representación y la especificación algebraica correspondiente fueron detalladas en el capítulo 2, por lo que aquí sólo se tratan los detalles relativos a su implementación mediante un paquete genérico y el programa de validación respectivo.

5.1 Implementación

A continuación se presentan comentarios sobre el código del programa del paquete genérico que introduce el tipo de dato abstracto "**colas con prioridades**" y lo implementa mediante apuntadores a nodos.

El paquete consta de dos partes: la especificación que funciona como interfaz para los usuarios y el cuerpo que contiene los detalles de la implementación. Para evitar una posible confusión entre la especificación algebraica del tipo de datos abstracto **colas con prioridades** y la especificación del paquete genérico **ColasConPrioridades**, la referencia a este último será siempre como la **interfaz** del paquete.

5.11 Interfaz del paquete genérico

La interfaz es el medio de comunicación con el usuario. Está constituida, básicamente, por dos partes: una parte pública (que es la única a la que tiene verdadero acceso el usuario) y otra privada (que es usada básicamente para anunciarle al compilador algo de lo que vendrá después).

La parte pública describe: las características de los servicios que se ofrecen en el paquete -a través de sus funciones y procedimientos-; el número y tipo de parámetros que se deben proporcionar para llamar una operación; las funciones y los procedimientos genéricos cuya implementación estará a cargo del usuario una vez que haya instanciado el paquete genérico (esto es, una vez que haya creado un ejemplar del paquete referido a un tipo de elemento específico; por ejemplo: cola con prioridades de enteros o cola con prioridades de registros).

La parte privada introduce la declaración de los tipos usados para implementar el tipo básico a exportar: **ColaPr**, aunque sin dar el detalle interno, el cual se aclarará dentro del cuerpo del paquete.

El código completo del paquete genérico **ColasConPrioridades** se puede consultar en el **Anexo B**. A continuación se esquematizará la interfaz para resaltar sus elementos más importantes, los cuales se comentarán después. Las etiquetas descriptivas se escriben en itálicas.

Especificación del carácter de la unidad:
generic

Parámetros formales genéricos:
Tipo: type **Elem** is private;
Subprograma: **with** function "<"(Izq ...);

Nombre paquete:
package ColasConPrioridades is

Parte Pública:

Tipo exportable: type **ColaPr** is limited private

Subprogramas exportables:

procedure **Vacia**(La_Cola : in out ColaPr);

...

function **Maximo**(La_Cola: in ColaPr) return Elem;

...

Excepciones exportables:

Capacidad_Excedida : exception;

...

Especificación del carácter de la sección:

private

Tipo: type **Nodo**;

...

5.111 Especificación del carácter de la unidad

La interfaz del paquete genérico, se introduce con la declaración de "**generic**", lo cual implica que se trata de una plantilla que, antes de poder usarse, deberá ser instanciada.

Para poder instanciar el paquete genérico, el programa de aplicación debe proporcionar los parámetros formales declarados en su interfaz.

5.112 Parámetros formales genéricos

Los parámetros formales genéricos son como "blancos" guardadores de espacio que serán sustituidos en el programa de aplicación por la definición de los parámetros reales.

El tipo para caracterizar a los **elementos** se declara en forma genérica como **privado**. Esto permite que el parámetro formal pueda ser correspondido por el tipo actual (definido en el programa de aplicación) que permita las operaciones predefinidas de asignación y prueba de igualdad.

Para determinar la prioridad, la cual define la posición en que se insertará el nuevo elemento, se sobrecarga el operador relacional "<". La definición de esta función genérica dependerá del tipo real asignado al elemento; su implementación corresponderá al usuario del paquete genérico.

5.113 Parte Pública

- **Tipo exportable**

El tipo que exporta el paquete es la denominación de toda una clase de componentes: el de colas con prioridades, llamado en forma corta **ColaPr**, tipo **privado limitado** que es el mecanismo proporcionado por Ada para encapsular. Como no hay ninguna operación

implícita definida para este tipo, se obliga a definir las propias operaciones evitando que los usuarios violen la abstracción valiéndose del conocimiento de la representación subyacente.

- **Subprogramas exportables**

Los subprogramas exportables corresponden a las operaciones aplicables definidas en la especificación algebraica, declaradas con la sintaxis de Ada.

```
procedure Vacia (La_Cola: in out ColaPr);
procedure Mete (El_Elemento: in Elem; A_La_Cola: in out ColaPr);
procedure Saca_Max (La_Cola: in out ColaPr);
function Maximo (La_Cola: in ColaPr) return Elem;
function Es_Vacia (La_Cola: in ColaPr) return Boolean;
```

Si se necesita que devuelva un resultado se emplea una función, de otra manera se utiliza un procedimiento. En ambos casos es indispensable indicar el nombre y tipo de los parámetros requeridos.

Se incluye la declaración del procedimiento **Iterar** (iterador) y del parámetro **Proceso** (parámetro del subprograma genérico cuya implementación debe proporcionarse posteriormente).

```
generic
        with procedure Proceso (El_Elemento : in  Elem);
    procedure Iterar (Sobre_la_Cola : in ColaPr);
```

El subprograma genérico **Iterar** puede considerarse como un modelo que define operaciones, pero que no puede solicitarse directamente porque el compilador no le ha asignado una dirección de memoria que pueda ser referenciada. El modelo genérico **Iterar** se convierte en un subprograma ejecutable real por medio del proceso denominado instanciación, lo cual ocurre en los programas de aplicación.

- **Excepciones exportables**

Las dos posibilidades de error detectadas quedan declaradas en la parte de la especificación como excepciones:

```
Capacidad_Excedida : exception;
Capacidad_Agotada : exception;
```

5.114 Parte privada

En la parte privada de la especificación, se introduce en forma incompleta la declaración del tipo **Nodo**, el cual será definido en la parte del cuerpo del paquete, con el fin de poder declarar **Estructura** como un apuntador que tiene acceso a **Nodo**. Por último se declara que el tipo privado limitado **ColaPr** es un registro cuyo primer componente es **Principio** y cuyo segundo componente es **Final**, ambos del tipo **Estructura**, ya mencionado.

```
type Nodo;
type Estructura is access Nodo;
type ColaPr  is
  record
       Principio  :  Estructura;
       Final   :  Estructura;
          end record;
```

6.12 Cuerpo del paquete genérico

El cuerpo del paquete existe sólo para implementar los servicios ofrecidos en la interfaz, describe los mecanismos que determinan que los subprogramas se comporten en la forma como lo hacen.

Además de los cuerpos de los subprogramas declarados en la interfaz, el cuerpo del paquete puede definir recursos que serán utilizados por los cuerpos de los subprogramas del propio paquete, pero que no están destinados al uso externo.

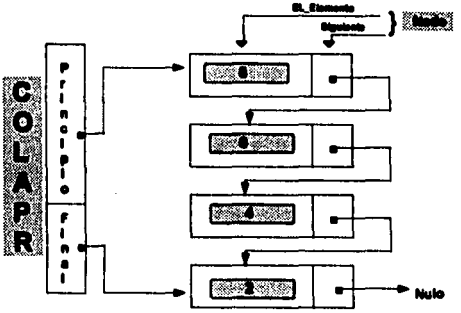
Los recursos que puede incluir son: declaraciones adicionales de variables, tipos y subprogramas. Tales facilidades son parte indispensable de la implementación del paquete pero resultan irrelevantes e inaccesibles para los usuarios externos.

5.121 Declaraciones internas

Dentro del cuerpo del paquete, aparece el tipo **Nodo**, introducido en forma incompleta en la parte privada de la interfaz, para ser definido dentro del cuerpo de la siguiente manera:

```
type Nodo is
  record
     El_Elemento  : Elem;
     Siguiente  : Estructura;
  end record;
```

El objeto cola con prioridades se constituye como una lista ligada (en la cual cada nodo contiene un elemento único) y un apuntador al nodo siguiente. De esta manera la cola con prioridades queda representada como el apuntador a un nodo que, a su vez, apunta a otros nodos.



Implementación de cola con prioridades

Debido a que esta representación se oculta dentro del cuerpo del paquete, se puede decir que cumple bien su papel porque permite al usuario manipular las colas mediante las operaciones exportadas, pero evita que abuse explotando la decisión de diseño tomada.

5.122 Operaciones

```
procedure Vacia (La_Cola : in out ColaPr) is
begin
  La_Cola := ColaPr'(Principio => null, Final  => null);
end Vacia;
```

En el momento de asignar una referencia nula a **Principio** y **Final**, crea una cola con prioridades vacía, eliminando cualquier elemento, si lo hubiera.

```
procedure Mete (El_Elemento : in  Elem;
          A_La_Cola : in out ColaPr) is
  Anterior  : Estructura;
  Apuntador : Estructura := A_La_Cola.Principio;
begin
  if A_La_Cola.Principio = null then
      A_La_Cola.Principio := new Nodo'(El_Elemento => El_Elemento,Siguiente => null);
      A_La_Cola.Final  := A_La_Cola.Principio;
  else  while (Apuntador /= null) and then  El_Elemento < Apuntador.El_Elemento loop
              Anterior := Apuntador;
              Apuntador := Apuntador.Siguiente;
        end loop;
        if Anterior = null then
              A_La_Cola.Principio := new Nodo'(El_Elemento => El_Elemento,
                                               Siguiente => Apuntador);
              if A_La_Cola.Final = null then  A_La_Cola.Final := A_La_Cola.Principio;
              end if;
        elsif Apuntador = null then
              A_La_Cola.Final.Siguiente := new Nodo'(El_Elemento => El_Elemento,
                                                     Siguiente => null);
              A_La_Cola.Final := A_La_Cola.Final.Siguiente;
        else  Anterior.Siguiente := new Nodo'(El_Elemento => El_Elemento,
                                              Siguiente => Apuntador);
        end if;
  end if;
exception
  when Storage_Error => raise Capacidad_Excedida;
end Mete;
```

Esta operación inserta un elemento considerando su prioridad. El elemento que ostente la mayor prioridad quedará al principio de la cola. Su determinación se hace con ayuda del subprograma declarado como parámetro genérico "<".

Primero se presenta el caso sencillo en el que la cola está vacía, en cuya ocurrencia simplemente se inserta el elemento. De otra manera se inicia la comparación de la prioridad del último elemento contra la del nuevo, para determinar cual debe quedar adelante.

Si se recorre toda la cola hasta el principio, significa que ningún otro elemento tuvo una prioridad mayor a la del nuevo elemento, por lo que éste será insertado al inicio.

El caso de activación automática de la excepción predefinida Storage_Error se encauza a través de una excepción específica definida como **Capacidad_Excedida**, para que pueda ser manejada convenientemente en el programa de aplicación.

```
procedure Saca_Max (La_Cola : in out ColaPr) is
begin
  La_Cola.Principio := La_Cola.Principio.Siguiente;
  if La_Cola.Principio = null then La_Cola.Final := null;
  end if;
exception
  when Constraint_Error => raise Capacidad_Agotada;
end Saca_Max;
```

Esta operación retira el elemento que está al principio el cual, en el caso de colas con prioridades, corresponde al que tiene mayor prioridad.

En lugar de verificar explícitamente si la cola está vacía se deja que la semántica de indirección de los apuntadores haga el trabajo. Esto significa que, si el apuntador llamado **Principio** de la cola que se recibe como parámetro apunta a **null** (lo cual quiere decir que no hay elemento alguno y que no puede desplazarse siquiera una vez) activará automáticamente la llamada de la excepción estándar definida con el nombre de **Constraint_Error**.

```
function Maximo (La_Cola : in ColaPr) return Elem is
begin
  return La_Cola.Principio.El_Elemento;
exception
  when Constraint_Error => raise Capacidad_Agotada;
end Maximo;
```

Esta operación selectora **Máximo** devuelve el elemento que está al principio el cual, en el caso de colas con prioridades, corresponde al que tiene mayor prioridad.

```
function Es_Vacia (La_Cola : in ColaPr) return Boolean is
begin
  return(La_Cola.Principio = null);
end Es_Vacia;
```

Es_Vacia devuelve verdadero cuando existe una referencia nula, lo cual implica que no hay elementos en la cola.

```
procedure Iterar (Sobre_la_Cola : in ColaPr) is
  El_Iterador : Estructura := Sobre_la_Cola.Principio;
begin
  while not (El_Iterador = null) loop
        Proceso(El_Iterador.El_Elemento);
        El_Iterador := El_Iterador.Siguiente;
  end loop;
end Iterar;
```

Visita cada elemento de la cola desde el frente de la cola, por lo que inicializa al iterador asignándole el valor que tiene en su elemento llamado Principio

A partir de ese punto aplica el procedimiento **Proceso** (que es el subprograma que el usuario del paquete debe implementar y pasar como parámetro al subprograma genérico **Iterar**, según puede apreciarse en el programa de validación, en el cual quedan concretados como **Despliega_Colores** y **Despliega_Reg**[1]) el cual permite desplegar o imprimir cada uno de los elementos de la estructura.

A continuación, se desplaza al siguiente elemento al hacer que **El_Iterador** apunte al elemento referido por su campo **Siguiente**, para repetir el ciclo en tanto no llegue al final de la cola (en cuyo caso el iterador apuntará a null), lo cual llevaría a terminar la ejecución del procedimiento.

5.2 Validación de la Implementación

El lenguaje Ada no tiene una especificación formal que pudiera permitir la demostración formal del programa que implementa el tipo de datos abstracto **colas con prioridades** frente a especificación algebraica correspondiente.

En estas circunstancias se decidió utilizar un tipo de prueba "de caja blanca" que, aprovechando el conocimiento de los detalles de la implementación hecha, pudiera llevar a probar todas las operaciones básicas y sus diferentes opciones, con la posibilidad de que fuera factible repetir las pruebas bajo las mismas condiciones u otras diferentes.

El programa utilizado para validar la implementación es un programa interactivo que permite seleccionar el tipo de colas con prioridades con el que se desea trabajar y la clase de operación que se desea efectuar (cualquiera de las operaciones definidas como válidas en la especificación algebraica correspondiente, más el despliegue del contenido de la cola o el regreso al menú de selección de tipo de colas).

Durante la ejecución del programa se pueden crear paulatinamente las colas deseadas, solicitando su despliegue posterior para comprobar el estado de la cola. Además, es posible generar errores que activen a los manejadores de excepciones previstos que, dependiendo del caso, permitirán corregir el error o, en caso de violaciones a la estructura de la cola (como por ejemplo, intentar sacar de una cola vacía), enviará un mensaje, saldrá de ese nivel y solamente permitirá seguir con una nueva sesión interactiva.

5. 21 Estructura del programa

El código completo del programa de validación de la implementación de colas con prioridades se puede consultar en el **Anexo B**. A continuación se esquematizará su estructura para destacar los elementos importantes los cuales se comentarán posteriormente. Las etiquetas descriptivas aparecen en cursivas.

Importaciones:
with...

Nombre del subprograma:
PRCOLPRI

[1] Ver punto 5.212 Parte declarativa del procedimiento de este mismo capítulo

```
Parte declarativa del subprograma:
    declaraciones de tipos:  type Colores is (Ambar, ...);
    declaraciones de variables:  elem_color:Colores;
    declaraciones de subprogramas:
            function "<"(Izq:in Alumno; ...)return ...;
    instanciaciones de paquetes genéricos:
            package ColaPrRegs is new ColasConPrioridad(...);
    instanciaciones de subprogramas genéricos:
            procedure Despliega_ColaRegs is new ColaPrRegs.Iterar(...);
    cuerpos de subprogramas:
            function "<"(Izq:in Alumno; ...) return ... is ...
      manejadores de excepciones del nivel:
            exception when ColasPrEnum.Capacidad_Agotada  ...
```

```
Cuerpo del subprograma:
    secuencia de comandos:
            Menu;
            Submenu;
            ...
manejadores de excepciones del nivel
            exception when Io_exceptions.Data_error => ...
            ...
```

El orden en el que aparecen los elementos citados depende de las necesidades de declaraciones posteriores, debido a que no se puede hacer mención a un objeto que no haya sido previamente introducido por medio de una declaración.

Un programa en Ada es simplemente un subprograma que puede ser de dos tipos: procedimiento o función. La única diferencia entre un programa principal y cualquier otro subprograma es que éste último no tiene que ser invocado mediante el enunciado de un procedimiento o la evaluación de una expresión. El programa principal es llamado por el sistema operativo cuando se pide su ejecución.

En el caso del programa de validación de la implementación de colas con prioridades, el programa principal se localiza en el cuerpo del procedimiento llamado **PRCOLPRI**, cuyos elementos se analizarán a continuación.

5.211 Importaciones

Lo primero que se establece son las dependencias del programa lo cual hace mediante la cláusula de contexto "**with**". Aquí aparece la relación con el paquete genérico que contiene la implementación del tipo de dato abstracto que interesa: **ColasConPrioridades.**

5.212 Parte declarativa del procedimiento

En esta parte aparecen las declaraciones, instanciaciones y subprogramas necesarios para que las instrucciones del programa principal se ejecuten adecuadamente. Se pueden identificar los siguientes elementos:

- Declaraciones de los dos tipos (uno primitivo y otro estructurado) con los que se va a trabajar:

```
type Colores is (Ambar, Blanco, Cafe, Gris, Negro);
type Alumno is record
        Cuenta : natural range 0..10 := 0;
        Apellido : string(1..10) := "          ";
        Nombre : string(1..10) := "          ";
end record;
```

- Declaraciones de variables, en este caso de paquetes ya instanciados:

```
ColaPr1 : ColasPrEnum.ColaPr;
ColaPr2 : ColaPrRegs.ColaPr;
```

- Declaraciones de subprogramas:

```
function "<" (Izquierdo : in Alumno;
     Derecho   : in Alumno) return Boolean;
```

que se sobrecarga con el fin de aplicarse al ejemplo de colas con registros, para el cual no está implícitamente definida.

- Instanciaciones de paquetes genéricos:

```
package Color_Io is new  Text_Io.Enumeration_Io(Colores);
package ColasPrEnum is new  ColasConPrioridades( Elem => Colores, "<" => "<" );
package ColaPrRegs  is new  ColasConPrioridades( Elem => Alumno, "<" => "<" );
```

- Instanciaciones de subprogramas genéricos:

```
procedure Despliega_ColasPrColores is new ColasPrEnum.Iterar(Proceso =>
     Despliega_Colores);
procedure Despliega_ColaPrRegs is new ColaPrRegs.Iterar(Proceso => Despliega_Reg);
```

Obsérvese que, en ambos casos de instanciaciones, se utiliza la palabra reservada "**new**" y se proporcionan los parámetros reales pertinentes. Nótese también que la llamada al procedimiento **Iterar** se hace con notación de punto referida a las instanciaciones de paquetes genéricos recién obtenidas.

- Cuerpos de subprogramas que se usarán dentro de la ejecución del programa:

```
procedure Despliega_Colores (El_Elemento : in Colores) is
begin
     New_Line;
     Color_Io.Put(El_Elemento);
end Despliega_Colores;
```

```
procedure Despliega_Reg (El_Elemento : in Alumno) is
begin
    New_Line;
    Put(El_Elemento.Cuenta); put(" ");
    Put(El_Elemento.Apellido); put (" ");
    Put(El_Elemento.Nombre);
end Despliega_Reg;
```

Estos dos procedimientos describen el despliegue de elementos específicos con parámetros reales. Obsérvese que constituyen los subprogramas concretos que se requieren como parámetros reales para crear las instancias correspondientes del procedimiento genérico **Iterar**, ya mencionadas anteriormente, las cuales nos permitirán recorrer (y desplegar) las colas con prioridades deseadas.

```
function "<" (Izquierdo : in Alumno; Derecho : in Alumno) return Boolean is
begin
 if Izquierdo.Apellido < Derecho.Apellido then return True;
 elsif Izquierdo.Apellido = Derecho.Apellido
    then return ( (Izquierdo.Nombre) < (Derecho.Nombre) );
    else return False;
 end if;
end "<";
```

La definición de la función de comparación se hace considerando cada uno de los componentes del registro.

En este grupo quedan también:

- El procedimiento **Menú** que propone las opciones disponibles para la selección del tipo de cola que se desea manipular.

- El procedimiento **Submenú** que presenta las opciones de operaciones disponibles para este tipo de datos abstractos y permite probar la implementación de cada una ellas. Es en este procedimiento donde pueden verse las diferentes llamadas que se hacen, según sea el tipo de cola y el tipo de operación solicitado. Por ejemplo, veamos el código correspondiente a la selección de la operación 1 para crear una cola con prioridades vacía:

```
when 1 =>           -- crear una cola con prioridades vacia
 case sel_cola is
    when 1 =>       ColasPrEnum.Vacia(ColaPr1);
    when 2 =>       ColaPrRegs.Vacia(ColaPr2);
    when others => null;
 end case;
```

Nótese que, el llamado a la operación **Vacía** se hace a través de notación de punto referida a la instanciación específica y con el parámetro del tipo correspondiente.

- Los manejadores de excepciones que se activan cuando el usuario pretende visualizar el elemento con prioridad máxima (función **Máximo**) o sacar el elemento con máxima prioridad (procedimiento **Saca_Max**) en ambos casos de una cola vacía. El tratamiento que se les dió aquí fue: desplegar un mensaje de aviso y regresar al usuario al Menú que muestra el tipo de colas, en los siguientes términos:

```
exception
   when ColasPrEnum.Capacidad_Agotada | ColaPrRegs.Capacidad_Agotada=>
      put(" ERROR: Tratas de sacar de una cola vacía. ");
      put("       Regresamos al menu de tipos de colas.");
```

5.213 Cuerpo del procedimiento

En el cuerpo del procedimiento reside lo que puede considerarse el programa principal de la validación.

Está formado por una serie de comandos que, en un ciclo, llaman sucesivamente a los procedimientos **Menú** y **Submenú**, hasta que cambia el valor de la condición que lleva a su fin al ciclo o se activa alguno de los manejadores de excepciones que se controlan en este nivel, el más externo del programa, cuya consecuencia es un mensaje y la salida inmediata del programa. Todo esto se expresa con el código que se cita a continuación:

```
   salirProgr := FALSE;
loop
   MENU;
   exit when salirProgr;
   SUBMENU;
endloop;
exception
    when lo_exceptions.Data_error =>
        put ("- - - Oprimiste tecla equivocada. ");
        put ("- - - Salimos del programa. ");
    when ColasPrEnum.Capacidad_Excedida
            | ColaPrRegs.Capacidad_Excedida =>
        put("Rebasamos capacidad de memoria. Salimos del programa. ");
```

5.22 Pruebas de validación

Dado que se trata de un programa interactivo, cada sesión puede conducirse de la manera que se desee. Sin embargo, el conjunto de pruebas que se efectuó, con el propósito de validar todas las operaciones implementadas en el módulo mediante los axiomas correspondientes, siguió una secuencia como la que se describe a continuación:

5.221 Pruebas que implican operaciones con colas vacías

1) Se escoge uno de los dos tipos de elementos que se ofrecen, por ejemplo el tipo primitivo Colores que es un tipo de enumeración para el cual existe intrínsecamente la relación de ordenamiento, lo cual se hace con la opción 1 del menú de validación del paquete genérico de colas con prioridades.

2) Se crea una cola vacía (opción 1 del submenú de operaciones sobre colas con prioridades).

3) Se intenta sacar el elemento con valor máximo de la cola (opción 3 del submenú), lo cual activará la excepción correspondiente definida como Capacidad_Agotada la cual, por ser considerada como un intento de violación a la naturaleza de la

estructura, tiene una penalización fuerte que, previo aviso, nos sacará del programa. Esto prueba el axioma 1: sacamax(vacia)=error.
Para reanudar las pruebas se debe invocar nuevamente el programa y repetir los pasos 1 y 2.

4) Se trata de ver el elemento con valor máximo de la cola (opción 4 del submenú), lo cual, al igual que en el punto 3, nos sacará del programa. Con esto se ve que se cumple el axioma 3: maximo(vacia)=error.
Para reanudar las pruebas se debe invocar nuevamente el programa y repetir los pasos 1 y 2.

5) Se pregunta si la cola con prioridades está vacía con la opción 5 del submenú que deberá desplegar un mensaje afirmativo el cual corroborará el cumplimiento del axioma 5: esvacia(vacia)=cierto.

5.222 Pruebas que implican operaciones con colas no vacías

6) Con la opción 2, se mete el elemento Blanco, que está dentro de la gama de posibilidades que ofrece el tipo Colores (Ámbar, Blanco, Café, Gris, Negro).

7) Se activa la opción 5 del submenú para ver si ahora está vacía la cola. Aparecerá un mensaje negativo que confirmará el cumplimiento del axioma 6:
esvacia(mete(e,c))=falso.

8) Se pide ver el elemento con valor máximo de la cola con prioridades (opción 4 del submenú). La respuesta será Blanco de conformidad con la primera parte del axioma 4: maximo(mete(e,c))=SI esvacia(c) o maximo(c)<e ENT e.

9) Se usa la opción 3 del submenú para sacar el elemento con valor máximo de la cola con prioridades. Como sólo hay un elemento, la cola se quedará vacía, de acuerdo con la primera parte del axioma 2:
sacamax(mete(e,c))=SI esvacia(c) ENT vacia.
Tal condición se puede probar desplegando la cola con la opción 6 del submenú o preguntando si está vacia a través de la opción 5.

10) Se repite la acción señalada en el punto 6, para tener una cola con un solo elemento: Blanco.

11) Se inserta el elemento Café con la opción 2. De acuerdo con la definición del tipo de enumeración, este elemento tiene valor 3, en tanto que Blanco ostenta el valor 2. De esta manera, el nuevo elemento debe ser insertado al principio de la cola, lo cual se puede comprobar al desplegar los elementos contenidos con la opción 6.

12) Con la opción 4 se pide ver el elemento con valor máximo de la cola. El valor desplegado deberá ser Café de acuerdo con la condición disyuntiva del axioma 4: maximo(mete(e,c))=SI ... o maximo(c)<e ENT e.

13) Se activa la opción 3 para sacar el elemento con valor máximo de la cola con prioridades e inmediatamente después la opción 6 para desplegar la cola resultante El elemento resultante deberá ser la cola original con un único elemento (Blanco), de acuerdo con lo establecido en la parte media del axioma 2:
sacamax(mete(e,c))= SI_NO SI maximo(c)<e ENT c.
Esto es así porque la operación sacamax de la cola original (Blanco) a la que se había insertado un nuevo elemento con mayor prioridad (Café) da por resultado la cola original.

14) Se repite la acción señalada en el punto 11 para tener una cola con dos elementos: (Café,Blanco).

15) Con la opción 2 del submenú, se introduce el elemento Ámbar con lo cual la cola tendrá tres elementos: (Café, Blanco, Ámbar). Se usa la opción 6 para corroborarlo.

16) Se aplica la opción 4 para ver el elemento con valor máximo, el cual deberá ser Café, de acuerdo con lo establecido en la parte final del axioma 4:
maximo(mete(e,c))=SI .. o ENT .. SI_NO maximo(c).

17) Se saca el elemento con valor máximo mediante la opción 3 y se despliega la cola restante con la opción 6. Los elementos que se desplegarán serán (Blanco y Ambar), lo cual coincide con la parte final del axioma 2:
sacamax(mete(e,c))=SI ... ENT ... SI_NO SI ... ENT .. SI_NO
mete(e,sacamax(c)).

Se podrían continuar las pruebas, metiendo y sacando nuevos elementos o repitiendo algunos de ellos. Sin embargo, con las pruebas hasta aquí comentadas se puede asegurar que se han explorado todas las posibilidades básicas establecidas en los axiomas.

5.23 Reuso para los otros módulos

Los programas de validación para los otros componentes fueron similares a éste.

En realidad se reusó la estructura ya creada para usarla a manera de patrón en el que se fueron haciendo las sustituciones necesarias para adecuarla al tipo de datos abstracto, al número y tipo de operaciones válidas, a los diferentes tipos de elementos primitivos o estructurados que se deseaba emplear, al llamado de funciones y procedimientos con los parámetros requeridos.

CONCLUSIONES

- En la creación de sistemas complejos de *software* es importante contar con un conjunto de principios y metodologías que garanticen la calidad del *software* resultante y faciliten su construcción. La *Ingeniería de Software* proporciona el marco de desarrollo adecuado.

- Dos ideas fundamentales de la *Ingeniería de Software*, modularidad y reusabilidad de componentes, son promovidas por las ideas aportadas por el paradigma orientado a objetos.

- La fase de diseño, con la especificación formal de las características que se esperan del producto final, es crucial como base sólida para un desarrollo conveniente del sistema de *software* porque permite contar con un esquema de comparación para su evaluación posterior.

- Los tipos de datos abstractos son concepciones que siempre están presentes en los desarrollos de software, por lo que su especificación formal resulta de gran utilidad en forma permanente.

- Ada es un lenguaje de programación moderno que tiene características que lo catalogan como lenguaje de programación basado en objetos y que ofrece estructuras, como los paquetes, que facilitan la modularidad y permiten la encapsulación, lo cual resulta ideal para implementar tipos de datos abstractos concebidos como módulos que agrupan un tipo de datos con el grupo de operaciones que le son aplicables, que cuentan con una interfaz que sirve como interlocutor con el usuario pero que, al mismo tiempo, le oculta los detalles de la implementación.

- Adicionalmente, Ada tiene la característica de genericidad, a través de los paquetes genéricos, la cual consiste en la posibilidad de describir patrones con detalles no especificados para hacerlos aplicables a situaciones abstractas que evitarán que se deba multiplicar el código similar aplicable a condiciones específicas.

Parecía que se conjuntaban las condiciones ideales para que se pudieran implementar los módulos correspondientes a los tipos de datos abstractos con cuyas especificaciones algebraicas ya se contaba. Sin embargo, se tuvieron algunas dificultades en su desarrollo como las que describen a continuación.

- La facilidad genérica de Ada es, en sí misma, una característica magnífica. Sin embargo, la versión estándar vigente no considera a los paquetes genéricos como tipos verdaderos, por lo que no pueden ser pasados como parámetros.
  Las consecuencias de esta situación son que, efectivamente, se cuenta con una implementación genérica, digamos de colas, que se puede instanciar para tener una implementación específica aplicable a colas de enteros, colas de caracteres, colas de cadenas, colas de registros, pero que de ninguna manera permite usar la especificación genérica para modificar o ampliar sus atributos y poder obtener una implementación específica de colas circulares u otra de prioridades, las cuales, posteriormente, fueran aplicadas como colas circulares de enteros, colas circulares de caracteres, o colas con prioridades de enteros, colas con prioridades de caracteres, etcétera.

- El estándar vigente de Ada carece del mecanismo de herencia que, en los lenguajes orientados a objetos facilita el reuso y extensión de los módulos.
  El nuevo estándar, conocido como Ada 9X, que está por aprobarse, incluye innovaciones entre las que se cuenta el manejo de herencia. Esto dará nuevas posibilidades a la implementación de los módulos reusables

- No se pudieron lograr implementaciones genéricas de los diferentes recorridos de los árboles. Solamente fue posible producirlas como parte de los programas de validación de los programas de implementación de árboles, una vez que se habían instanciado los árboles correspondientes a tipos específicos.

- A pesar de que Ada cuenta con la especificación de un medio ambiente mínimo (MAPSE) que debe garantizar una cierta facilidad a los programadores, la versión del compilador disponible fue totalmente primitivo. No contaba siquiera con un editor asociado que facilitara la corrección y prueba del código producido o que permitiera consultar los módulos y las operaciones que le eran relativas.

- Asimismo, la versión del compilador disponible establece que la interfaz y el cuerpo de un paquete genérico deben constituir una unidad de compilación. Al tener que aparecer juntos, rompen el principio de encapsulación porque exponen los detalles de la implementación al conocimiento del usuario y hacen factible el uso y un eventual abuso de ellos.

- Por tales razones, se dejaron sin implementar algunas de las especificaciones algebraicas existentes.

- Por ahora, la presente biblioteca queda como un esfuerzo de implementación que podrá servir de base para las producciones futuras, con versiones más adecuadas al uso que se pretende.

ANEXO A: BIBLIOTECA DE COMPONENTES REUSABLES

1. LISTAS

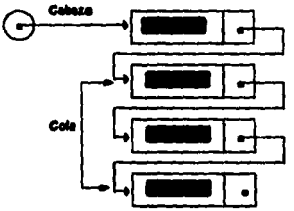
1.1 Definición

Una lista es una secuencia lineal de cero hasta un número arbitrario de elementos del mismo tipo, en la cual los elementos pueden ser insertados o removidos prácticamente de cualquier posición.

Se puede pensar en ejemplos como una lista de empleados, una lista de precios de artículos a la venta, una lista de nuevas publicaciones. Sus usos en computación son abundantes y muy variados, apareciendo en otras estructuras como colas, anillos, conjuntos, árboles y gráficas y en aplicaciones de bases de datos, sistemas operativos o inteligencia artificial.

1.2 Diagrama de representación

Una lista sin elementos es una lista nula. El primer elemento de la lista es llamado cabeza de lista y el resto, que puede ser desde cero hasta más elementos, es denominado como cola de la lista.



Lista

1.3 Especificación algebraica

```
ESQUEMA ListasEsq
|
    PARAM Elementos;
        EXPORTA TODO;
        GENERO Elem;
    FIN DE PARAM Elementos;
|;

ESPEC Listas;
    IMPORTA TODO DE Elementos;
            TODO DE Booleanos;
    EXPORTA TODO;
    GENERO Lista;
    OPERACIONES
        [OP1] vacia: → Lista;
        [OP2] cons:Elem*Lista→ Lista;
        [OP3] resto:Lista → Lista;
        [OP4] esvacia:Lista → Bool;
        [OP5] primero:Lista. → Elem;
        [OP6] concat:Lista*Lista → Lista
    VAR e:Elem, l, l1: Lista;
    AXIOMAS
        [L1] resto(vacia)=error
        [L2] resto(cons(e,l))=l
        [L3] esvacia(vacia)=cierto
        [L4] esvacia(cons(e,l))=falso
        [L5] primero(vacia)=error
        [L6] primero(cons(e,l))=e
        [L7] concat(vacia,l1)=l1
        [L8] concat(cons(e,l),l1)=cons(e,concat(l,l1))
FIN DE ESPEC Listas;
FIN DE ESQ ListasEsq;
```

1.4 Paquete genérico Listas

1.41 Interfaz

```
    with Text_IO;
generic
    type Elem is private;

package Listas is

    type Lista is private;
    procedure Vacia  (La_Lista  : in out Lista);
    procedure Cons (El_Elemento : in  Elem; Y_La_Lista : in out Lista);
    function Resto (La_Lista : in Lista) return Lista;
    function Es_Vacia (La_Lista : in Lista) return Boolean;
    function Primero (La_Lista : in Lista) return Elem;
    procedure Concatena (La_Lista1 : in out Lista; Con_La_Lista2 : in out Lista);

    Capacidad_Excedida : exception;
    La_Lista_Esta_Vacia : exception;

private
    type Nodo;
    type Lista is access Nodo;

end Listas;
```

1.42 Cuerpo

```
package body Listas is

type Nodo is
    record
        El_Elemento   Elem;
        Siguiente : Lista;
    end record;
    -- El objeto lista se representa como apuntador a un nodo que a su vez apunta a otros nodos.

procedure Vacia (La_Lista : in out Lista) is
begin
    La_Lista := null;
end Vacia;

procedure Cons (El_Elemento : in Elem; Y_La_Lista: in out Lista) is
begin
    Y_La_Lista := new Nodo'(El_Elemento => El_Elemento,
                            Siguiente => Y_La_Lista);
exception
    when Storage_Error => raise Capacidad_Excedida;
```

```
end Cons;

function Resto (La_Lista : in Lista) return Lista is
begin
      return La_Lista.Siguiente;
      -- Siguiente apunta a la lista que existía antes de que se agregara el último elemento.
exception
      when Constraint_Error => raise La_Lista_Esta_Vacia;
end Resto;

function Es_Vacia (La_Lista : in Lista) return Boolean is
begin
      return (La_Lista = null);
end Es_Vacia;

function Primero (La_Lista : in Lista) return Elem is
begin
      return La_Lista.El_Elemento;
exception
      when Constraint_Error => raise La_Lista_Esta_Vacia;
end Primero;

procedure Concatena (La_Lista1 : in out Lista; Con_La_Lista2 : in out Lista) is
      Indice_Lista1 : Lista := La_Lista1;
      Indice_Temporal : Lista;
begin
      if La_Lista1 = null then
            La_Lista1 := Con_La_Lista2;
            Con_La_Lista2 := null;
      else
            while Indice_Lista1.Siguiente /= null loop
                  Indice_Lista1 := Indice_Lista1.Siguiente;
            end loop;          -- Llegamos al último elemento de lista1
            Indice_Temporal := Indice_Lista1.Siguiente;
            Indice_Lista1.Siguiente := Con_La_Lista2;
            Con_La_Lista2 := Indice_Temporal;
       end if;
end Concatena;

end Listas;
```

2. LISTAS GENERALES

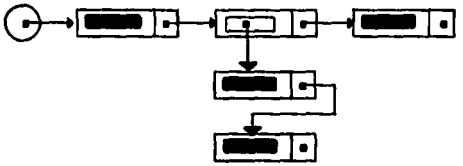
2.1 Definición

Una lista general es una secuencia lineal de cero hasta un número arbitrario de elementos del mismo tipo, en la cual los elementos pueden ser insertados o removidos prácticamente de cualquier posición en la que, al estilo del lenguaje de programación Lisp, los componentes pueden ser atómicos (es decir ser elementos indivisibles) o ser, a su vez, otras listas.

Se puede pensar en ejemplos como una lista de empleados, una lista de precios de artículos a la venta, una lista de nuevas publicaciones en las cuales los componentes pueden ser elementos individuales o ser, a su vez, otras listas, en cuyo caso se les denomina sublistas.

Como en el caso de listas, sus usos en computación son abundantes, apareciendo en otras estructuras como colas, anillos, conjuntos, árboles y gráficas y en aplicaciones de bases de datos, sistemas operativos o inteligencia artificial, pero a la vez son más versátiles porque permiten el manejo de listas de listas.

2.2 Diagrama de representación



LISTAS GENERALES

2.3 Especificación algebraica

```
ESQUEMA ListasGeneralesEsq
[
    PARAM Elementos;
        EXPORTA TODO;
        GENERO Elem;
    FIN DE PARAM Elementos;
];

ESPEC ListasGenerales;
    IMPORTA TODO DE Elementos;
            TODO DE Booleanos;
    EXPORTA TODO;
    GENERO ListaG;
    OPERACIONES
        [OP1] vacia: → ListaG;
        [OP2] cons:Elem*ListaG → ListaG;
        [OP3] conslista:ListaG*ListaG → ListaG;
        [OP4] resto:ListaG → ListaG;
        [OP5] esvacia:ListaG → Bool;
        [OP6] esprimelem:ListaG→Bool;
        [OP7] primelem:ListaG → Elem;
        [OP8] primlista:ListaG → ListaG;
    VAR e:Elem, l,l1 :Lista;
    AXIOMAS
        [LG1]  resto(vacia)=error
        [LG2]  resto(cons(e,l))=l
        [LG3]  resto(conslista(l,l1))=l1
        [LG4]  esvacia(vacia)=cierto
        [LG5]  esvacia(cons(e,l))=falso
        [LG6]  esvacia(conslista(l,l1))=falso
        [LG7]  esprimelem(vacia)=falso
        [LG8]  esprimelem(cons(e,l))=cierto
        [LG9]  esprimelem(conslista(l,l1))=falso
        [LG10] primelem(vacia)=error
        [LG11] primelem(cons(e,l))=e
        [LG12] primelem(conslista(l,l1))=error
        [LG13] primlista(vacia)=error
        [LG14] primlista(cons(e,l))=error
        [LG15] primlista(cons(l,l1))=l
FIN DE ListasGenerales;
FIN DE ESQ ListasGeneralesEsq;
```

2.4 Paquete genérico ListasGenerales

2.41 Interfaz

```
with Text_Io; use Text_Io;

generic
  type Elem is private;

package ListasGenerales is

  type Lista_G is private;
  Lista_Vacia : constant Lista_G;
  procedure Vacia (La_Lista : in out Lista_G);
  procedure Cons (El_Elemento: in Elem; Y_La_Lista : in out Lista_G);
  procedure Cons_Lista (La_Lista1 : in Lista_G; Y_La_Lista2 : in out Lista_G);
  function Resto (La_Lista : in Lista_G) return Lista_G;
  function Es_Vacia (La_Lista : in Lista_G) return Boolean;
  function Es_Prim_Elem (La_Lista : in Lista_G) return Boolean;
  function Prim_Elem (La_Lista : in Lista_G) return Elem;
  function Prim_Lista (La_Lista : in Lista_G) return Lista_G;

  Capacidad_Excedida : exception;
  La_Lista_Esta_Vacia : exception;

private
  type Seleccion is (Elemento,Sublista);
  type Nodo (Variante : Seleccion);
  type Lista_G is access Nodo;
  Lista_Vacia : constant Lista_G := null;

end ListasGenerales;
```

3.42 Cuerpo

```
package body ListasGenerales is

type Nodo (Variante : Seleccion) is
    record
        Siguiente : Lista_G;
        case Variante is
            when Elemento => El_Elemento : Elem;
            when Sublista => La_Sublista : Lista_G;
        end case;
    end record;
    -- El objeto Lista_G se representa como apuntador a un nodo que a su vez apunta a otros nodos, los
    --cuales distinguen cuando un componente es un elemento o una sublista.

procedure Vacia     (La_Lista : in out Lista_G) is
begin
    La_Lista := null;
end Vacia;
```

```
procedure Cons (El_Elemento : in  Elem;  Y_La_Lista : in out Lista_G) is
begin
      Y_La_Lista := new Nodo'(Variante => Elemento,
                              El_Elemento => El_Elemento,
                              Siguiente => Y_La_Lista);
exception
      when Storage_Error => raise Capacidad_Excedida;
end Cons;

procedure Cons_Lista (La_Lista1: in Lista_G;  Y_La_Lista2 : in out Lista_G) is
begin
      Y_La_Lista2 := new Nodo'(Variante => Sublista,
                               La_Sublista => La_Lista1,
                               Siguiente => Y_La_Lista2);
exception
      when Storage_Error  => raise Capacidad_Excedida;
end Cons_Lista;

function Resto (La_Lista : in Lista_G)  return Lista_G is
begin
      return La_Lista.Siguiente;
exception
      when Constraint_Error => raise La_Lista_Esta_Vacia;
end Resto;

function Es_Vacia (La_Lista  : in Lista_G)  return Boolean is
begin
      return (La_Lista  = null);
end Es_Vacia;

function Es_Prim_Elem (La_Lista : in Lista_G) return Boolean is
begin
      return (La_Lista.Variante = Elemento);
end Es_Prim_Elem;

function Prim_Elem (La_Lista : in Lista_G)  return Elem is
begin
      if La_Lista.Variante = Elemento then  return La_Lista.El_Elemento;
      end if;
exception
      when Constraint_Error => raise La_Lista_Esta_Vacia;
end Prim_Elem;

function  Prim_Lista(La_Lista : in Lista_G) return Lista_G is
begin
      if La_Lista.Variante = Sublista then return La_Lista.La_Sublista;
      end if;
exception
      when Constraint_Error => raise La_Lista_Esta_Vacia;
end Prim_Lista;

end ListasGenerales;
```

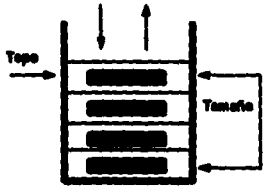
3. PILAS

A. 31 Definición

Los datos del tipo pilas se definen como secuencias finitas de elementos de cualquier tipo sobre las cuales es posible realizar la operación de meter un nuevo elemento o sacar un elemento de ella observando siempre la siguiente propiedad: el último elemento que se metió a la pila es siempre el primer elemento que se va a sacar. Por sus siglas, esta propiedad es conocida como UEPS (Último en Entrar, Primero en Salir).

Los ejemplos de aplicaciones dados para el caso de pilas son muy sencillos: una pila de platos o una pila de productos en repisas de mucho fondo, cuyo último elemento será, por razones prácticas, el primero en ser retirado. En computación su uso es esencial en todo compilador, sistema operativo, editor de textos y en muchas otras aplicaciones.

3.2 Diagrama de representación



La pila

3.3 Especificación algebraica

```
ESQUEMA PilasEsq
[
    PARAM Elementos;
        EXPORTA TODO;
        GENERO Elem;
    FIN DE PARAM Elementos;
]:

ESPEC Pilas;
    IMPORTA TODO DE Elementos;
        TODO DE Booleanos;
    EXPORTA TODO;
    GENERO Pila;
    OPERACIONES
        [OP1] vacia: → Pila;
        [OP2] mete:Elem *Pila → Pila;
        [OP3] saca:Pila → Pila;
        [OP4] esvacia:Pila → Bool;
        [OP5] tope:Pila → Elem;
    VAR e:Elem, p:Pila;
    AXIOMAS
        [P1] saca(vacia)=error
        [P2] saca(mete(e,p))=p
        [P3] esvacia(vacia)=cierto
        [P4] esvacia(mete(e,p))=falso
        [P5] tope(vacia)=error
        [P6] tope(mete(e,p))=e
FIN DE Pilas;
FIN DE ESQ PilasEsq;
```

3.4 Paquete genérico Pilas

3.41 Interfaz

```
with Text_IO; use Text_IO;
generic
  type Elemento is private;

package PILAS is

  type Pila is limited private;
  procedure Pila_Vacia (La_Pila : in out Pila);
  procedure Mete (El_Elemento : in Elemento; En_La_Pila : in out Pila);
  procedure Saca (La_Pila : in out Pila);
  function Es_Vacia (La_Pila : in Pila) return Boolean;
  function Tope (La_Pila : in Pila) return Elemento;

  -- Declaraciones para el iterador pasivo
  generic
        with procedure Proceso (El_Elemento : in Elemento);
  procedure Iterar (Sobre_la_Pila : in Pila);

  Saca_De_Pila_Vacia : exception;
  Tope_De_Pila_Vacia : exception;

private
  type Nodo;
  type Pila is access Nodo;

end PILAS;
```

3.42 Cuerpo

```
package body PILAS is

type Nodo is
      record
            El_Elemento : Elemento;
            Siguiente   : Pila;
      end record;
      -- El objeto pila se representa como apuntador a un nodo que a su vez apunta a otros nodos.

procedure Pila_Vacia (La_Pila : in out Pila) is
 begin
      La_Pila := null;
 end Pila_Vacia;
```

```
procedure Mete (El_Elemento  : in Elemento; En_La_Pila : in out Pila) is
 begin
     En_La_Pila := new Nodo'(El_Elemento => El_Elemento, Siguiente  => En_La_Pila);
end Mete;

procedure Saca (La_Pila : in out Pila) is
 begin
     La_Pila := La_Pila.Siguiente;
 exception
     when Constraint_Error => raise Saca_De_Pila_Vacia;
 end Saca;

function Es_Vacia (La_Pila : in Pila) return Boolean is
 begin
     return (La_Pila = null);
 end Es_Vacia;

function Tope (La_Pila : in Pila) return Elemento is
 begin
     return La_Pila.El_Elemento;
 exception
     when Constraint_Error => raise Tope_De_Pila_Vacia;
 end Tope;

procedure Iterar (Sobre_la_Pila : in Pila) is
     El_Iterador : Pila := Sobre_la_Pila
 begin
     while not (El_Iterador = null) loop
          Proceso(El_Iterador.El_Elemento
          El_Iterador := El_Iterador.Siguiente;
     end loop;
 end Iterar;

end PILAS;
```

4. PILAS ACOTADAS

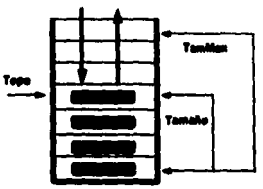
4.1 Definición

Los datos del tipo pilas se definen como secuencias finitas de elementos de cualquier tipo sobre las cuales es posible realizar la operación de meter un nuevo elemento o sacar un elemento de ella observando siempre la siguiente propiedad: el último elemento que se metió a la pila es siempre el primer elemento que se va a sacar. Por sus siglas, esta propiedad es conocida como UEPS (Último en Entrar, Primero en Salir).

En este caso, existe una limitante al tamaño máximo que puede llegar a tener la pila, el cual debe darse como un parámetro. Tal situación resulta útil cuando se tiene una capacidad de memoria limitada.

Los ejemplos de aplicaciones dados para el caso de pilas son muy sencillos: una pila de platos o una pila de productos en repisas de mucho fondo, cuyo último elemento será, por razones prácticas, el primero en ser retirado. En computación su uso es esencial en todo compilador, sistema operativo, editor de textos y en muchas otras aplicaciones.

4.2 Diagrama de representación



Pila acotada

4.3 Especificación algebraica

```
ESQUEMA PilasAcotadasEsq
[
    PARAM Elementos;
        EXPORTA TODO;
        GENERO Elem;
    FIN DE PARAM Elementos;

    PARAM TamMax;
        IMPORTA Nat DE Naturales;
        EXPORTA TODO,
        OPERACION
            Max: → Nat;
    FIN DE PARAM Tam_Max;
];

INSTANCIA PilasEsq;
    CON Elementos COMO Elementos,
        Elem COMO Elem;
FIN DE INSTANCIA PilasEsq;

ESPEC PilasAcotadas;
    IMPORTA TODO DE Elementos;
            TODO DE TamMax;
            Nat, +1, _=_ DE Naturales;
            TODO DE Pilas;
    EXPORTA Pila, vacia, mete, saca, esvacia, tope DE Pilas;
            esllena DE PilasAcotadas;
    OPERACIONES
        [OP1] esllena:Pila → Bool;
        [OP2] tamaño:Pila → Nat
        VAR e:Elem, p:pila;
    VAR e:Elem, p:Pila;

    AXIOMAS
        [PA7] tamaño(vacia)=0
        [PA8] tamaño(mete(e,p))=tamaño(p)+1
        [PA9] es_llena(p)=SI tamaño(p)=Max ENT cierto
                          SI_NO falso
        [PA10] mete(e,p)=SI es_llena(p) ENT error
                         SI_NO mete(e,p)
FIN DE PilasAcotadas;
FIN DE ESQ PilasAcotadasEsq;
```

4.4 Paquete genérico PilasAcotadas

4.41 Interfaz

```
      with Text_IO; use Text_IO;
generic
      type Elemento is private;

package PilasAcotadas is

      type Pila (Tam_Max : Positive) is limited private;
      procedure Pila_Vacia (La_Pila : in out Pila);
      procedure Mete (El_Elemento : in  Elemento; En_La_Pila : in out Pila);
      procedure Saca  (La_Pila : in out Pila);
      function Es_Vacia (La_Pila : in Pila) return Boolean;
      function Tope (La_Pila : in Pila) return Elemento;
      function Es_Llena (La_Pila : in Pila) return Boolean;
      function Tamano (La_Pila : in Pila) return Natural;

      Mete_En_Pila_Llena : exception;
      Saca_De_Pila_Vacia : exception;
      Tope_De_Pila_Vacia : exception;

private
      type Elem is array(Positive range <>) of Elemento;
      type Pila(Tam_Max : Positive) is
            record
                  El_Tope : Natural;
                  Los_Elementos : Elem(1 .. Tam_Max);
            end record;
end PilasAcotadas;
```

4.42 Cuerpo

```
package body PilasAcotadas is

procedure Pila_Vacia  (La_Pila : in out Pila) is
begin
      La_Pila.El_Tope  := 0;            -- tamaño(pila vacia)=0
end Pila_Vacia;

procedure Mete (El_Elemento : in Elemento;   En_La_Pila : in out Pila) is
begin
      En_La_Pila.Los_Elementos(En_La_Pila.El_Tope + 1) := El_Elemento;
      En_La_Pila.El_Tope := En_La_Pila.El_Tope + 1;
exception
      when Constraint_Error => raise Mete_En_Pila_Llena;
end Mete;
```

```
procedure Saca (La_Pila : in out Pila) is
begin
      La_Pila.El_Tope := La_Pila.El_Tope - 1;
exception
      when Constraint_Error => raise Saca_De_Pila_Vacia;
end Saca;

function Es_Vacia (La_Pila : in Pila) return Boolean is
begin
      return (La_Pila.El_Tope  = 0);
end Es_Vacia;

function Tope  (La_Pila : in Pila) return Elemento is
begin
      return La_Pila. Los_Elementos(La_Pila.El_Tope);
exception
      when Constraint_Error => --raise Tope_De_Pila_Vacia;
end Tope;

function Es_Llena (La_Pila : in Pila) return Boolean is
begin
      return(La_Pila.El_Tope = La_Pila.Tam_Max);
end Es_Llena;

function Tamano (La_Pila : in Pila) return Natural is
begin
      return La_Pila.El_Tope;
end Tamano;

end PilasAcotadas;
```

5. COLAS

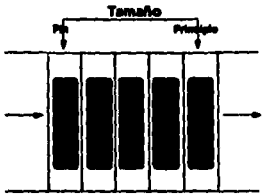
5.1 Definición

Una cola es una secuencia lineal de un número arbitrario de elementos del mismo tipo en la cual estos son incorporados por un lado y removidos por el otro.

Debido a que generalmente las inserciones se hacen por el frente y los retiros por el final, se dice que las colas constituyen un tipo de dato abstracto con estructura PEPS (Primeras Entradas, Primeras Salidas).

En la vida diaria existen mucha aplicaciones para este tipo como la fila de atención a clientes frente a una caja de cobro o frente a una taquilla de boletos, el vuelo en círculo de aviones en espera de pista, o la asignación de recursos que realiza el sistema operativo de un sistema multiusuario respecto a impresoras, accesos a disco, tiempo de procesador, etc.

5.2 Diagrama de representación



Cola

5.3 Especificación algebraica

```
ESQUEMA ColasEsq;
[
    PARAM Elementos
    EXPORTA TODO;
    GENERO Elem;
    FIN DE PARAM  Elementos;
];
ESPEC Colas;
    IMPORTA TODO DE Elementos;
            TODO DE Booleanos,
    EXPORTA TODO;
    GENERO Cola;
    OPERACIONES
        [OP1] vacia: → Cola;
        [OP2] meteult: Elem*Cola→ Cola;
        [OP3] sacaprim: Cola→Cola;
        [OP4] primero: Cola → Elem;
        [OP5] esvacia: Cola → Bool;
    VAR c:Cola, e:Elem;
    AXIOMAS
        [C1] sacaprim(vacia)=error
        [C2] sacaprim(meteult(e,c))=SI esvacia(c) ENT c
                                    SI_NO meteult(e,sacaprim(c))
        [C3] primero(vacia)=error
        [C4] primero(meteult(e,c))=SI esvacia(c) ENT e
                                    SI_NO primero(c)
        [C5] esvacia(vacia)=cierto
        [C6] esvacia(meteult(e,c))=falso
FIN DE Colas;
FIN DE ESQ ColasEsq;
```

5.4 Paquete genérico Colas

5.41 Interfaz

```
generic
  type Elem is private;

package Colas is

  type Cola is limited private;
  procedure Vacia (La_Cola : in out Cola);
  procedure Mete_Ult (El_Elemento : in Elem;  A_La_Cola : in out Cola);
  procedure Saca_Prim (La_Cola : in out Cola);
  function Primero (La_Cola : in Cola) return Elem;
  function Es_Vacia (La_Cola : in Cola) return Boolean;

  -- Declaraciones para el iterador pasivo:
  generic
       with procedure Proceso (El_Elemento : in  Elem);
  procedure Iterar (Sobre_la_Cola : in Cola);

  Capacidad_Excedida : exception;
  Capacidad_Agotada : exception;

private
  type Nodo;
  type Estructura is access Nodo;
  type Cola is
       record
            Principio  :  Estructura;
            Final   :  Estructura;
       end record;

end Colas;
```

5.42 Cuerpo

```
package body Colas is

type Nodo is
     record
          El_Elemento : Elem;
          Siguiente  : Estructura;
     end record;

procedure Vacia (La_Cola : in out Cola) is
begin
     La_Cola := Cola'(Principio => null,
                      Final  => null);
     -- La cola está vacía cuando Principio y Final son nulos.
```

```
end Vacia;

procedure Mete_Ult (El_Elemento : in Elem;
      A_La_Cola : in out Cola) is
begin
      if A_La_Cola.Principio = null then A_La_Cola.Principio := new Nodo'(El_Elemento =>
El_Elemento,
                                                                  Siguiente => null);
                                          A_La_Cola.Final  := A_La_Cola.Principio;
      else  A_La_Cola.Final.Siguiente := new Nodo'(El_Elemento => El_Elemento, Siguiente => null);
            A_La_Cola.Final := A_La_Cola.Final.Siguiente;
      end if;
exception
      when Storage_Error => raise Capacidad_Excedida;
end Mete_Ult;

procedure Saca_Prim (La_Cola : in out Cola) is
begin
      La_Cola.Principio := La_Cola.Principio.Siguiente;
      if La_Cola.Principio = null then  La_Cola.Final := null;
      end if;
exception
      when Constraint_Error => raise Capacidad_Agotada;
end Saca_Prim;

function Primero (La_Cola : in Cola) return Elem is
begin
      return La_Cola.Principio.El_Elemento;
exception
      when Constraint_Error => raise Capacidad_Agotada;
end Primero;

function Es_Vacia (La_Cola : in Cola) return Boolean is
begin
      return(La_Cola.Principio = null);
end Es_Vacia;

procedure Iterar (Sobre_la_Cola : in Cola) is
      El_Iterador : Estructura := Sobre_la_Cola.Principio;
begin
      while not (El_Iterador = null) loop
            Proceso(El_Iterador.El_Elemento);
            El_Iterador := El_Iterador.Siguiente;
      end loop;
end Iterar;

end Colas;
```

6. COLAS ACOTADAS

6.1 Definición

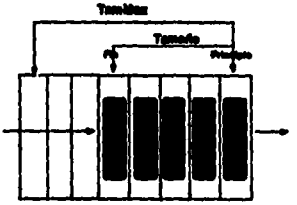
Una cola es una secuencia lineal de un número arbitrario de elementos del mismo tipo en la cual estos son incorporados por un lado y removidos por el otro.

Debido a que generalmente las inserciones se hacen por el frente y los retiros por el final, se dice que las colas constituyen un tipo de dato abstracto con estructura PEPS (Primeras Entradas, Primeras Salidas).

En este caso, existe una limitación referente al tamaño máximo que puede tener la cola, dato que se dará como un parámetro. Tal situación es útil cuando la capacidad de memoria puede llegar a ser un problema.

En la vida diaria existen mucha aplicaciones para este tipo como la fila de atención a clientes frente a una caja de cobro o frente a una taquilla de boletos, el vuelo en círculo de aviones en espera de pista, o la asignación de recursos que realiza el sistema operativo de un sistema multiusuario respecto a impresoras, accesos a disco, tiempo de procesador, etc.

6.2 Diagrama de representación



Cola acotada

6.3 Especificación algebraica

```
ESQUEMA ColasAcotadasEsq
[
    PARAM Elementos;
        EXPORTA TODO:
        GENERO Elem;
    FIN DE PARAM Elementos;
    PARAM TamMax;
        IMPORTA Nat DE Naturales;
        EXPORTA TODO;
     OPERACION
        Max: → Nat;
    FIN DE PARAM TamMax;
];

INSTANCIA ColasEsq;
    CON Elementos COMO Elementos,
        Elem COMO Elem,
FIN DE INSTANCIA ColasEsq;

ESPEC ColasAcotadas;
    IMPORTA TODO DE Elementos;
                TODO DE TamMax;
                Nat,_+1,_=_ DE Naturales;
                TODO DE Colas;
    EXPORTA Cola,vacia, meteult, sacaprim, primero, esvacia DE Colas;
                esllena DE ColasAcotadas;
    OPERACIONES
        [OP1] esllena:Cola → Bool;
        [OP2] tamaño:Cola → Nat,
    VAR c:Cola, e:Elem;
AXIOMAS
    [CA7] tamaño(vacia)=0
    [CA8] tamaño(meteult(e,c))=tamaño(c)+1
    [CA9] esllena(c)=SI tamaño(c)=Max ENT cierto
                    SI_NO falso
    [CA10] meteult(e,c)=SI es_llena(c) ENT error
                        SI_NO meteult(e,c)
FIN DE ColasAcotadas;
FIN DE ESQ ColasAcotadasEsq,
```

6.4 Paquete genérico ColasAcotadas

6.41 Interfaz

```
generic
  type Elem is private;

package ColasAcotadas is

  type Cola(Tam_Max : Positive) is limited private;
  procedure Vacia (La_Cola : in out Cola);
  procedure Mete_Ult (El_Elemento : in  Elem;  A_La_Cola : in out Cola);
  procedure Saca_Prim (La_Cola : in out Cola);
  function Primero (La_Cola : in Cola) return Elem;
  function Es_Vacia (La_Cola : in Cola) return Boolean;
  function Es_Llena (La_Cola : in Cola) return Boolean;
  function Tamano (La_Cola : in Cola) return Natural;

  Capacidad_Excedida : exception;
  Capacidad_Agotada : exception;

private
  type Elems is array(Positive range <>) of Elem;
  type Cola(Tam_Max : Positive) is
       record
            El_Fondo : Natural := 0;
            Los_Elems : Elems(1 .. Tam_Max);
       end record;

end ColasAcotadas;
```

6.42 Cuerpo

```
package body ColasAcotadas is

procedure Vacia (La_Cola : in out Cola) is
begin
     La_Cola.El_Fondo := 0;
     -- El valor asociado a la cola vacía es 0.
end Vacia;

procedure Mete_Ult (El_Elemento : in Elem;  A_La_Cola : in out Cola) is
begin
     A_La_Cola.Los_Elems(A_La_Cola.El_Fondo + 1) := El_Elemento;
     A_La_Cola.El_Fondo := A_La_Cola.El_Fondo + 1;
exception
     when Constraint_Error => raise Capacidad_Excedida;
end Mete_Ult;
```

```
procedure Saca_Prim (La_Cola : in out Cola) is
begin
    if La_Cola.El_Fondo = 0 then  raise Capacidad_Agotada;
    elsif La_Cola.El_Fondo = 1 then  La_Cola.El_Fondo := 0;
    else
        La_Cola.Los_Elems(1 .. (La_Cola.El_Fondo - 1)) :=
                                          La_Cola.Los_Elems(2 .. La_Cola.El_Fondo);
        La_Cola.El_Fondo := La_Cola.El_Fondo - 1;
    end if;
end Saca_Prim;

function Primero (La_Cola : in Cola) return Elem is
begin
    if La_Cola.El_Fondo = 0 then  raise Capacidad_Agotada;
    else  return La_Cola.Los_Elems(1);
    end if;
end Primero;

function Es_Vacia (La_Cola : in Cola) return Boolean is
begin
    return(La_Cola.El_Fondo = 0);
end Es_Vacia;

function Es_Llena (La_Cola    : in Cola) return Boolean is
begin
    return(La_Cola.El_Fondo = Tam_Max);
end Es_Llena;

function Tamano (La_Cola : in Cola) return Natural is
begin
    return La_Cola.El_Fondo;
end Tamano;

end ColasAcotadas;
```

7. BICOLAS

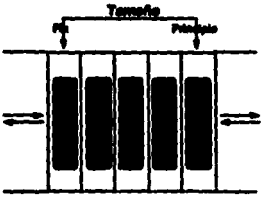
7.1 Definición

Una bicola es una secuencia lineal de un número arbitrario de elementos del mismo tipo en la cual estos pueden ser incorporados y removidos por un lado o por el otro, indistintamente.

En cierta forma el comportamiento de una bicola no obedece el comportamiento PEPS (Primeras Entradas, Primeras Salidas) atribuido a las colas.

Las aplicaciones que las usan generalmente lo que aprovechan es la facilidad para poder insertar elementos en el frente de la cola.

7.2 Diagrama de representación



Bicola

7.3 Especificación algebraica

```
ESQUEMA BiColasEsq
{
    PARAM Elementos;
    EXPORTA TODO;
    GENERO Elem;
FIN DE PARAM Elementos;
};

ESPEC BiColas;
    IMPORTA TODO DE Elementos;
            TODO DE Booleanos.
    EXPORTA TODO;
    GENERO BiCola;
    OPERACIONES
        [OP1] vacia: → BiCola
        [OP2] meteizq:Elem*BiCola → BiCola;
        [OP3] meteder:Elem*BiCola → BiCola;
        [OP4] sacaizq:BiCola → BiCola;
        [OP5] sacader:BiCola → BiCola;
        [OP6] izq:BiCola → Elem;
        [OP7] der:BiCola → Elem;
        [OP8] esvacia:BiCola → Bool;
    VAR c:BiCola, e:Elem;
    AXIOMAS
        [BC1]  sacaizq(vacia)=error
        [BC2]  sacaizq(meteizq(e,c))=c
        [BC3]  sacaizq(meteder(e,c))=SI esvacia(c) ENT vacia
                                    SI_NO meteder(e,sacaizq(c))
        [BC4]  sacader(vacia)=error
        [BC5]  sacader(meteizq(e,c))=SI esvacia(c) ENT vacia
                                    SI_NO meteizq(e,sacader(c))
        [BC6]  sacader(meteder(e,c))=c
        [BC7]  izq(vacia)=error
        [BC8]  izq(meteizq(e,c))=e
        [BC9]  izq(meteder(e,c))=SI esvacia(c) ENT e
                                 SI_NO izq(c)
        [BC10] der(vacia)=error
        [BC11] der(meteizq(e,c))=SI esvacia(c) ENT e
                                 SI_NO der(c)
        [BC12] der(meteder(e,c))=e
        [BC13] esvacia(vacia)=cierto
        [BC14] esvacia(meteizq(e,c))=falso
        [BC15] esvacia(meteder(e,c))=falso
FIN DE BiColas;
FIN DE ESQ BiColasEsq;
```

7.4 Paquete genérico Bicolas

7.41 Interfaz

```
generic
    type Elem is private;

package Bicolas is

    type Bicola is limited private;
    type Lado is (Lado_Izq, Lado_Der);
    procedure Vacia (La_Bicola : in out Bicola);
    procedure Mete(El_Elemento : in Elem;  A_La_Bicola : in out
Bicola;
                   En_Lado : in Lado);
    procedure Saca (La_Bicola : in out Bicola; En_Lado : in Lado);
    function Izq (La_Bicola : in Bicola) return Elem;
    function Der (La_Bicola :  in Bicola) return Elem;
    function Tamano (La_Bicola : in Bicola) return Natural;
    function Es_Vacia (La_Bicola : in Bicola) return Boolean;

    -- Declarac para el iterador pasivo
    generic
        with procedure Proceso (El_Elemento : in Elem);
    procedure Iterar (Sobre_La_Bicola : in Bicola);

    Capacidad_Excedida : exception;
    Capacidad_Agotada : exception;

private
    type Nodo;
    type Estructura is access Nodo;
    type Bicola is
        record
            Izquierda : Estructura;
            Derecha : Estructura;
        end record;

end Bicolas;
```

7.42 Cuerpo

```
package body Bicolas is

type Nodo is
    record
        El_Elemento: Elem;
        Siguiente: Estructura;
    end record;
```

```
                                    Derecha => null);
    -- La Bicola está vacía cuando Izquierda  y Derecha son nulos.
end Vacia;

procedure Mete (El_Elemento : in Elem;  A_La_Bicola : in out Bicola;  En_Lado in   Lado) is
begin
    if A_La_Bicola.Izquierda  = null then
                A_La_Bicola.Izquierda  := new Nodo'(El_Elemento => El_Elemento,  Siguiente =>
null);
                A_La_Bicola.Derecha  := A_La_Bicola.Izquierda ;
    else
     if En_Lado = Lado_Izq then
                A_La_Bicola.Izquierda  := new Nodo'(El_Elemento => El_Elemento,
                                                    Siguiente  => A_La_Bicola.Izquierda );
    else
        A_La_Bicola.Derecha.Siguiente := new Nodo'(El_Elemento => El_Elemento,
                                                    Siguiente   => null);
        A_La_Bicola.Derecha := A_La_Bicola.Derecha.Siguiente;
     end if;
  end if;
exception
    when Storage_Error => raise Capacidad_Excedida;
end Mete;

procedure Saca (La_Bicola : in out Bicola;  En_Lado : in  Lado) is
Indice : Estructura := La_Bicola.Izquierda;
begin
    -- Primero verificamos para ver si sólo hay un elemento en la bicola.
    -- Si es así, simplemente lo eliminamos.
    --Como efecto lateral se activará un error si la cola está vacía.
    if Indice.Siguiente = null then
        La_Bicola.Izquierda := null;
        La_Bicola.Derecha   := null;
    -- Si estamos sacando un elemento de la izquierda, sólo avanzamos Izquierda al nodo que le sigue.
    elsif En_Lado = Lado_Izq then
        La_Bicola.Izquierda := La_Bicola.Izquierda.Siguiente;
    -- En el otro caso, debemos recorrer la lista desde la izquierda hasta que Indice apunte al nodo que
    -- preceda al que apunta Derecha. En este momento, desechamos el elemento apuntado por Derecha y
    -- hacemos que éste apunte ahora al nodo que lo precede.
    else
        while Indice.Siguiente /= La_Bicola.Derecha loop
            Indice := Indice.Siguiente;
        end loop;
        La_Bicola.Derecha := Indice;
        La_Bicola.Derecha.Siguiente := null;
    end if;
exception
    -- en lugar de verificar explícitamente si la cola está vacía, dejamos que la semántica de indirección
de
    -- los apuntadores haga el trabajo
    when Constraint_Error => raise Capacidad_Agotada;
```

```
end Saca;

function Izq (La_Bicola : in Bicola) return Elem is
begin
      return La_Bicola.Izquierda .El_Elemento;
exception
      -- en lugar de verificar explícitamente si la cola está vacía dejamos que la semántica de indirección
de
      -- los apuntadores haga el trabajo
      when Constraint_Error => raise Capacidad_Agotada;
end Izq;

function Der(La_Bicola : in Bicola) return Elem is
begin
      return La_Bicola.Derecha.El_Elemento;
exception
      when Constraint_Error => raise Capacidad_Agotada;
end Der;

function Tamano (La_Bicola : in Bicola) return Natural is
      Contador : Natural  := 0;
      Apuntador : Estructura := La_Bicola.Izquierda ;
begin
      while Apuntador /= null loop
            Contador := Contador + 1;
            Apuntador := Apuntador.Siguiente;
      end loop;
      return Contador;
end Tamano;

function Es_Vacia (La_Bicola : in Bicola) return Boolean is
begin
      return(La_Bicola.Izquierda  = null);
end Es_Vacia;

procedure Iterar (Sobre_La_Bicola : in Bicola) is
      El_Iterador : Estructura := Sobre_La_Bicola.Izquierda;
begin
      while not (El_Iterador = null) loop
            Proceso(El_Iterador.El_Elemento);
            El_Iterador := El_Iterador.Siguiente;
      end loop;
end Iterar;

end Bicolas;
```

8. COLAS CON PRIORIDADES

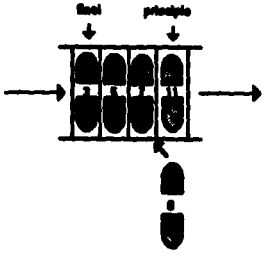
8.1 Definición

Una cola es una secuencia lineal de un número arbitrario de elementos del mismo tipo en la cual estos son incorporados por un lado y removidos por el otro.

En la cola con prioridades, la prioridad de un elemento afecta el orden en el cual éste será atendido; entre mayor sea su prioridad más pronto será retirado de la cola.

En la vida diaria existen muchas aplicaciones para este tipo como la asignación de pista a aviones con problemas de combustible o de funcionamiento, o la asignación de recursos que realiza el sistema operativo de un sistema multiusuario respecto a impresoras, accesos a disco, tiempo de procesador, etcétera, a usuarios que ostentan distintos grados de preferencia de atención.

8.2 Diagrama de representación



8.3 Especificación algebraica

ESQUEMA ColasConPrioridadesEsq

```
|
    PARAM ElementosOrd;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO;
        GENERO Elem;
        OPERACIONES
            _=_ :Elem*Elem →Bool;
            _<_ :Elem*Elem → Bool;
        VAR e1, e2, e3: Elem;
        AXIOMAS
            (e1=e1)=cierto
            (e1=e2)=(e2=e1)
            (e1=e2) y (e2=e3) ⇒ (e1=e3)=cierto
            (e1=e2) ⇒ (e1<e2)=falso
            (e1<e2) y (e2<e3) ⇒ (e1<e3)=cierto
    FIN DE PARAM ElementosOrd;
|
ESPEC ColasConPrioridades;
    IMPORTA TODO DE ElementosOrd;
    EXPORTA TODO;
    GENERO ColaPr;
    OPERACIONES
        [OP1] vacia: → ColaPr;
        [OP2] mete:Elem*ColaPr → ColaPr;
        [OP3] sacamax:ColaPr → ColaPr;
        [OP4] maximo:ColaPr → Elem;
        [OP5] esvacia:ColaPr → Bool;
    VAR c:ColaPr, e:Elem;
    AXIOMAS
        [CP1] sacamax(vacia)=error
        [CP2] sacamax(mete(e,c))= SI esvacia(c) ENT vacia
                                  SI_NO SI maximo(c)<e ENT c
                                        SI_NO mete(e,saca max(c))
        [CP3] maximo(vacia)=error
        [CP4] maximo(mete(e,c))=SI esvacia(c) o maximo(c)<e ENT e
                                SI_NO maximo(c)
        [CP5] esvacia(vacia)=cierto
        [CP6] esvacia(mete(e,c))=falso
FIN DE ESPEC ColasConPrioridades;
FIN DE ESQ ColasConPrioridadesEsq
```

8.4 Paquete genérico ColasConPrioridades

8.41 Interfaz

```
generic
    type Elem is private;
    with function "<"  (Izquierdo : in Elem; Derecho  : in Elem) return Boolean;

package ColasConPrioridades is
    type ColaPr  is  limited private;
    procedure Vacia (La_Cola : in out ColaPr);
    procedure Mete (El_Elemento : in  Elem; A_La_Cola : in out ColaPr);
    procedure Saca_Max (La_Cola : in out ColaPr);
    function Maximo  (La_Cola : in ColaPr) return Elem;
    function Es_Vacia (La_Cola : in ColaPr) return Boolean;

    generic
   with procedure Proceso (El_Elemento : in  Elem);
    procedure Iterar (Sobre_la_Cola : in ColaPr);

    Capacidad_Excedida : exception;
    Capacidad_Agotada : exception;

private
    type Nodo;
    type Estructura is access Nodo;
    type ColaPr  is
      record
         Principio  :  Estructura;
         Final    :  Estructura;
      end record;

end ColasCon Prioridades;
```

8.42 Cuerpo

```
package body ColasConPrioridades is

type Nodo is
    record
        El_Elemento : Elem;
        Siguiente : Estructura;
    end record;

procedure Vacia (La_Cola : in out ColaPr) is
begin
    La_Cola := ColaPr'(Principio => null,
                       Final  => null);
    -- La cola está vacía cuando Principio y Final son nulos.
end Vacia;
```

```
procedure Mete (El_Elemento : in Elem;
                A_La_Cola  : in out ColaPr) is
    Anterior  : Estructura;
    Apuntador : Estructura := A_La_Cola.Principio;
begin
    if A_La_Cola.Principio = null then
        A_La_Cola.Principio := new Nodo'(El_Elemento => El_Elemento,
                                         Siguiente   => null);
        A_La_Cola.Final  := A_La_Cola.Principio;
    else
        while (Apuntador /= null) and then El_Elemento < Apuntador.El_Elemento loop
            Anterior := Apuntador;
            Apuntador := Apuntador.Siguiente;
        end loop;
        if Anterior = null then
            A_La_Cola.Principio := new Nodo'(El_Elemento => El_Elemento,
                                             Siguiente   => Apuntador);
            if A_La_Cola.Final = null then
                A_La_Cola.Final := A_La_Cola.Principio;
            end if;
        elsif Apuntador = null then
            A_La_Cola.Final.Siguiente := new Nodo'(El_Elemento => El_Elemento,
                                                   Siguiente   => null);
            A_La_Cola.Final := A_La_Cola.Final.Siguiente;
        else
            Anterior.Siguiente := new Nodo'(El_Elemento => El_Elemento,
                                            Siguiente => Apuntador);
        end if;
    end if;
exception
    when Storage_Error => raise Capacidad_Excedida;
end Mete;

procedure Saca_Max (La_Cola : in out ColaPr) is
begin
    La_Cola.Principio := La_Cola.Principio.Siguiente;
    if La_Cola.Principio = null then
        La_Cola.Final := null;
    end if;
exception
    when Constraint_Error => raise Capacidad_Agotada;
end Saca_Max;

function Maximo (La_Cola : in ColaPr) return Elem is
begin
    return La_Cola.Principio.El_Elemento;
exception
    when Constraint_Error => raise Capacidad_Agotada;
end Maximo;
```

```
function Es_Vacia (La_Cola : in ColaPr) return Boolean is
begin
    return(La_Cola.Principio = null);
end Es_Vacia;

function Tamano (La_Cola : in ColaPr) return Natural is
    Contador : Natural  := 0;
    Apuntador : Estructura := La_Cola.Principio;
begin
    while Apuntador /= null loop
        Contador := Contador + 1;
        Apuntador := Apuntador.Siguiente;
    end loop;
    return Contador;
end Tamano;

procedure Iterar (Sobre_la_Cola : in ColaPr) is
    El_Iterador : Estructura := Sobre_la_Cola.Principio;
begin
    while not (El_Iterador = null) loop
        Proceso(El_Iterador.El_Elemento);
        El_Iterador := El_Iterador.Siguiente;
    end loop;
end Iterar;

end ColasConPrioridades;
```

9. ÁRBOLES BINARIOS

9.1 Definición

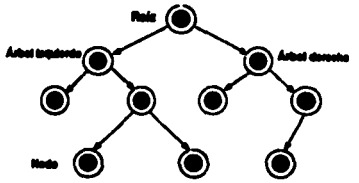
Un **árbol** es una colección de nodos que pueden tener un número arbitrario de referencias a otros nodos. Los árboles son estructuras no lineales que resultan muy útiles porque permiten representar una jerarquía entre los elementos de un conjunto.

En el mundo real existen muchos objetos que exteriorizan una estructura tipo árbol genérico como: los organigramas, los árboles genealógicos, los índices de libros, las guías de discursos.

El tipo de dato abstracto **árbol binario** se define como un conjunto finito de nodos, el cual puede ser vacío o constar de un elemento de datos (al que se da el nombre de raíz) y dos árboles binarios disjuntos (llamados subárbol izquierdo y subárbol derecho de la raíz). Las operaciones que incluye corresponden a las estrictamente necesarias para construir un árbol básico.

9.2 Diagrama de representación

Para cada par de nodos existe un único camino que los conecta. Si un árbol dado tiene cero nodos se dice que es un árbol nulo. En el caso de los **árboles binarios**, el número máximo de hijos que puede tener cada nodo es 2.



Árbol binario

9.3 Especificación algebraica

ESQUEMA Arboles BinariosEsq;

[

PARAM Elementos;

EXPORTA TODO;

GENERO Elem;

FIN DE PARAM Elementos;

];

ESPEC ArbolesBinarios;

IMPORTA TODO DE Elementos;

IMPORTA TODO DE Booleanos;

EXPORTA TODO;

GENERO ArbolBin;

OPERACIONES

[OP1] nulo: → ArbolBin;

[OP2] creanodo:ArbolBin*Elem*ArbolBin → ArbolBin;

[OP3] izq:ArbolBin → ArbolBin;

[OP4] der:ArbolBin → ArbolBin;

[OP5] elem:ArbolBin → Elem;

[OP6] esvacio:ArbolBin → Bool;

VAR ai,ad:ArbolBin, e:Elem;

AXIOMAS

[AB1] izq (nulo)=error

[AB2] izq(creanodo(ai,e,ad))=ai

AB3] der(nulo)error

[AB4] der(creanodo(ai,e,ad))=ad

[AB5] elem(nulo)=error

[AB6] elem(creanodo(ai,e,ad))=e

[AB7] esnulo(nulo)=cierto

[AB8] esnulo(creanodo(ai,e,ad))=falso

FIN DE ArbolesBinarios;

FIN DE ESQ ArbolesBinariosEsq;

9.4 Paquete genérico ArbolesBinarios

9.41 Interfaz

```
generic
    type Elem is private;

package ArbolesBinarios is

    type ArbolBin is private;
    Arbol_Vacio : constant ArbolBin;
    procedure Vacio (El_Arbol : in out ArbolBin);
    procedure CreaNodo (El_Elemento : in Elem; El_Subarbol_Izq : in ArbolBin;
                        El_Subarbol_Der : in ArbolBin; El_Arbol : in out ArbolBin);
    function Izq (El_Arbol : in ArbolBin) return ArbolBin;
    function Der (El_Arbol : in ArbolBin) return ArbolBin;
    function Elemento (El_Arbol : in ArbolBin) return Elem;
    function Es_Vacio (El_Arbol : in ArbolBin) return Boolean;

    Capacidad_Excedida : exception;
    Arbol_Es_Vacio : exception;

private
    type Nodo;
    type ArbolBin is access Nodo;
    Arbol_Vacio : constant ArbolBin := null;

end ArbolesBinarios;
```

9.42 Cuerpo

```
package body ArbolesBinarios is

    type Nodo is
        record
            El_Elemento : Elem;
            SubArbol_Izq : ArbolBin;
            SubArbol_Der : ArbolBin;
        end record;

procedure Vacio (El_Arbol : in out ArbolBin) is
begin
    El_Arbol := null;
end Vacio;
```

```
procedure CreaNodo (El_Elemento : in Elem;  El_Subarbol_Izq : in ArbolBin;
                    El_Subarbol_Der : in ArbolBin;  El_Arbol : in out ArbolBin) is
begin
    El_Arbol := new Nodo'(El_Elemento => El_Elemento,
                          SubArbol_Izq => El_Subarbol_Izq,
                          ubArbol_Der => El_Subarbol_Der);
exception
    when storage_Error => raise Capacidad_Excedida;
end CreaNodo;

function Izq (El_Arbol : in ArbolBin) return ArbolBin is
begin
    return El_Arbol.SubArbol_Izq;
exception
    when Constraint_Error => raise Arbol_Es_Vacio;
end Izq;

function Der (El_Arbol : in ArbolBin) return ArbolBin is
begin
    return El_Arbol.SubArbol_Der;
exception
    when Constraint_Error => raise Arbol_Es_Vacio;
end Der;

function Elemento  (El_Arbol : in ArbolBin) return Elem is
begin
    return El_Arbol.El_Elemento;
exception
    when Constraint_Error =>  raise Arbol_Es_Vacio;
end Elemento;

function Es_Vacio (El_Arbol : in ArbolBin) return Boolean is
begin
    return (El_Arbol = null);
end Es_Vacio;

end ArbolesBinarios;
```

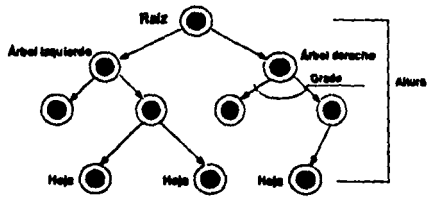
10. ÁRBOLES BINARIOS EXTENDIDOS

10.1 Definición

El **árbol binario** básico es un conjunto finito de nodos, el cual puede ser vacío o constar de un elemento de datos (al que se da el nombre de raíz) y dos árboles binarios disjuntos (llamados subárbol izquierdo y subárbol derecho de la raíz).

El tipo de dato abstracto **árbol binario extendido** incluye los conceptos de hoja, altura, camino mínimo, árbol completo, por lo que sus operaciones permiten determinar si un nodo es hoja, si un árbol es el último de su jerarquía, cuántos nodos tiene el árbol y cuántos de ellos son hojas, cuál es la altura del árbol, cuál es el camino mínimo con el que se puede recorrer y si se trata de un árbol completo o perfectamente balanceado.

10.2 Diagrama de representación



Árbol binario extendido

10.3 Especificación algebraica

```
ESQUEMA ArbolesBinariosExtEsq;
[
    PARAM Elementos;
        EXPORTA TODO;
        GENERO Elem;
    FIN DE PARAM Elementos;
],

INSTANCIA ArbolesBinariosEsq;
    CON Elementos COMO Elementos,
        Elem COMO Elem;
FIN DE INSTANCIA ArbolesBinariosEsq;

ESPEC ArbolesBinariosExt;
    IMPORTA TODO DE Elementos;
            TODO DE ArbolesBinarios;
            TODO DE Naturales;
    EXPORTA TODO DE ArbolesBinarios;
            esHoja, esUltimo, numNodos, altura, caminoMin,
            esCompleto, numHojas DE ArbolesBinariosExt;
    OPERACIONES
        [OP7]  esHoja:ArbolBin → Bool;
        [OP8]  esUltimo:ArbolBin → Bool;
        [OP9]  numNodos:ArbolBin→ Nat;
        [OP10] altura:ArbolBin → Nat;
        [OP10] caminoMin:ArbolBin → Nat;
        [OP12] esCompleto:ArbolBin→ Bool;
        [OP13] numHojas:ArbolBin→ Nat;
    VAR a,ai,ad:ArbolBin, e:Elem;
    AXIOMAS
        [AB9]  esHoja(vacio)=falso
        [AB10] esHoja(creaNodo(ai,e,ad))=(esNulo(ai) y esNulo(ad))
        [AB11] esUltimo(nulo)=falso
        [AB12] esUltimo(creaNodo(ai,e,ad))=(esNulo(ai) o esNulo(ad))
        [AB13] numNodos(nulo)=0
        [AB14] numNodos(creaNodo(ai,e,ad))=1+numNodos(ai)+numNodos(ad)
        [AB15] altura(nulo)=0
        [AB16] altura(creaNodo(ai,e,ad))=1+max(altura(ai), altura(ad))
        [AB17] caminoMin(nulo)=0
        [AB18] caminoMin(creaNodo(ai,e,ad))=1+min(caminoMin(ai), caminoMin(ad))
        [AB19] esCompleto(a)=(altura(a)=caminoMin(a))
        [AB20] numHojas(nulo)=0
        [AB21] numHojas(creaNodo(ai,e,ad))=SI esHoja(creaNodo(ai,e,ad)) ENT 1
                                           SI_NO numHojas(ai)+numHojas(ad)
FIN DE ArbolesBinariosExt;
FIN DE ESQ ArbolesBinariosExtEsq;
```

10.4 Paquete genérico ARBOLESBINARIOSEXT

10.41 Interfaz

```
generic
  type Elem is private;

package ARBOLESBINARIOSEXT is

  type ArbolBinX is private;
  Arbol_Vacio : constant ArbolBinX;
  procedure Vacio (El_Arbol : in out ArbolBinX);
  function CreaNodo (El_Elemento : in Elem; El_Subarbol_Izq : in ArbolBinX;
                     El_Subarbol_Der : in ArbolBinX) return ArbolBinX;
  function Izq (El_Arbol : in ArbolBinX) return ArbolBinX;
  function Der (El_Arbol : in ArbolBinX) return ArbolBinX;
  function Elemento (El_Arbol : in ArbolBinX) return Elem;
  function Es_Vacio (El_Arbol : in ArbolBinX) return Boolean;
  function Es_Hoja (El_Arbol : in ArbolBinX) return Boolean;
  function Es_Ultimo (El_Arbol : in ArbolBinX) return Boolean;
  function Num_Nodos (El_Arbol : in ArbolBinX) return Natural;
  function Altura (El_Arbol : in ArbolBinX) return Natural;
  function Camino_Min (El_Arbol : in ArbolBinX) return Natural;
  function Es_Completo (El_Arbol : in ArbolBinX) return Boolean;
  function Num_Hojas (El_Arbol : in ArbolBinX) return Natural;

  Capacidad_Excedida : exception;
  Arbol_Es_Vacio : exception;

private
  type Nodo;
  type ArbolBinX is access Nodo;
  Arbol_Vacio : constant ArbolBinX := null;

end ARBOLESBINARIOSEXT;
```

10.42 Cuerpo

```
package body ARBOLESBINARIOSEXT is

    type Nodo is
        record
            El_Elemento : Elem;
            SubArbol_Izq : ArbolBinX;
            SubArbol_Der : ArbolBinX;
        end record;

procedure Vacio (El_Arbol : in out ArbolBinX) is
begin
    El_Arbol := null;
end Vacio;
```

```
function CreaNodo (El_Elemento : in Elem; El_Subarbol_Izq : in  ArbolBinX;
                   El_Subarbol_Der : in  ArbolBinX) return ArbolBinX is
Nuevo_Nodo : ArbolBinX;
begin
     Nuevo_Nodo := new Nodo'(El_Elemento  => El_Elemento,
                             SubArbol_Izq => El_Subarbol_Izq,
                             SubArbol_Der => El_Subarbol_Der);
     return Nuevo_Nodo;
exception
     when storage_Error =>  raise Capacidad_Excedida;
end CreaNodo;

function Izq (El_Arbol : in ArbolBinX) return ArbolBinX is
begin
     return El_Arbol.SubArbol_Izq;
exception
     when Constraint_Error => raise Arbol_Es_Vacio;
end Izq;

function Der (El_Arbol : in ArbolBinX) return ArbolBinX is
begin
     return El_Arbol.SubArbol_Der;
exception
     when Constraint_Error => raise Arbol_Es_Vacio;
end Der;

function Elemento (El_Arbol : in ArbolBinX) return Elem is
begin
     return El_Arbol.El_Elemento;
exception
     when Constraint_Error =>  raise Arbol_Es_Vacio;
end Elemento;

function Es_Vacio (El_Arbol : in ArbolBinX) return Boolean is
begin
     return (El_Arbol = null);
end Es_Vacio;

function Es_Hoja (El_Arbol : in ArbolBinX) return Boolean is
begin
     if El_Arbol = null then
          return FALSE;
     else
          return (El_Arbol.SubArbol_Izq = null  and  El_Arbol.SubArbol_Der = null);
     end if;
end Es_Hoja;

function Es_Ultimo (El_Arbol : in ArbolBinX) return Boolean is
begin
     if El_Arbol = null then
          return FALSE;
     else
          return (El_Arbol.SubArbol_Izq = null  or  El_Arbol.SubArbol_Der = null);
     end if;
end Es_Ultimo;
```

```
function Num_Nodos (El_Arbol : in ArbolBinX) return Natural is
begin
    if El_Arbol = null then
        return 0;
    else
        return (1 + Num_Nodos(El_Arbol.SubArbol_Izq) + Num_Nodos(El_Arbol.SubArbol_Der));
    end if;
end Num_Nodos;

function Altura (El_Arbol : in ArbolBinX) return Natural is
    function Max (Alt_Arb1, Alt_Arb2 : Natural) return Natural is
    begin
        if Alt_Arb1 > Alt_Arb2 then return Alt_Arb1;
        else return Alt_Arb2;
        end if;
    end Max;
begin
    if El_Arbol = null then return 0;
    else return (1 + Max( Altura(El_Arbol.SubArbol_Izq), Altura(El_Arbol.SubArbol_Der)));
    end if;
end Altura;

function Camino_Min (El_Arbol : in ArbolBinX) return Natural is
    function Min (Alt_Arb1, Alt_Arb2 : Natural) return Natural is
    begin
        if Alt_Arb1 < Alt_Arb2 then return Alt_Arb1;
        else return Alt_Arb2;
        end if;
    end Min;
begin
    if El_Arbol = null then return 0;
    else return (1 + Min( Altura(El_Arbol.SubArbol_Izq), Altura(El_Arbol.SubArbol_Der)));
    end if;
end Camino_Min;

function Es_Completo(El_Arbol : in ArbolBinX) return Boolean is
begin
    return(Altura (El_Arbol) = Camino_Min(El_Arbol));
end Es_Completo;

function Num_Hojas (El_Arbol : in ArbolBinX) return Natural is
begin
    if El_Arbol = null then return 0;
    elsif (El_Arbol.SubArbol_Izq = null and El_Arbol.SubArbol_Der = null) then return 1;
    else
        return(Num_Hojas(El_Arbol.SubArbol_Izq)
            + Num_Hojas(El_Arbol.SubArbol_Der));
    end if;
end Num_Hojas;

end ARBOLESBINARIOSEXT;
```

11. ÁRBOLES DE BÚSQUEDA BINARIA

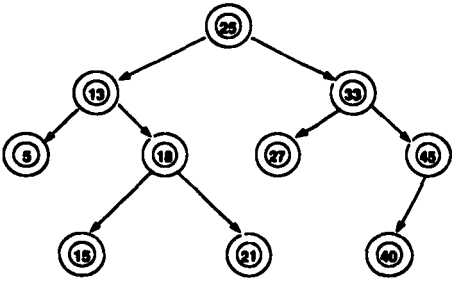
11.1 Definición

El tipo de dato abstracto **árbol de búsqueda binaria** es un árbol binario en el que los nodos están ordenados. El orden impuesto es tal que, para todo nodo, los valores de su subárbol izquierdo son menores que el valor en ese nodo y los valores en su subárbol derecho son mayores que él.

En computación, los árboles de búsqueda binaria se emplean en la búsqueda de soluciones de inteligencia artificial, las recuperaciones de bases de datos, la traducción del lenguaje natural, la representación de programas, el diseño de juegos, etcétera.

11.2 Diagrama de representación

Se esquematiza el caso de un árbol de búsqueda de enteros.



Árbol de búsqueda binaria

11.3 Especificación algebraica

```
ESQUEMA ArbolesDeBusquedaBinariaEsq;
{
    PARAM ElementosOrd;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO;
        GENERO Elem;
        OPERACION
            _=_:Elem*Elem → Bool;
            _<_:Elem*Elem → Bool;
        VAR e1,e2,e3:elem;
        AXIOMAS
            (e1=e1)=cierto
            (e1=e2)=(e2=e1)
            (e1=e2) y (e2=e3) ⇒ (e1=e3)=cierto
            (e1=e2) ⇒ (e1<e2)=falso
            (e1<e2) y (e2<e3) ⇒ (e1<e3)=cierto
    FIN DE PARAM ElementosOrd;
};

INSTANCIA ArbolesBinariosEsq;
    RENOMBRA ArbolBin COMO ArbolBB;
    CON ElementosOrd COMO Elementos,
        Elem COMO Elem;
FIN DE INSTANCIA ArbolesBinariosEsq;

ESPEC ArbolesDeBusquedaBinaria;
    IMPORTA TODO DE ElementosOrd;
            TODO DE ArbolesBinarios;
    EXPORTA ArbolBB, nulo DE ArbolesBinarios;
            inserta, sustrae, miembro, min DE ArbolesDeBusquedaBinaria;
    OPERACIONES
        [OP7] inserta:Elem*ArbolBB → ArbolBB;
        [OP8] sustrae:Elem*ArbolBB → ArbolBB;
        [OP9] esmiembro:Elem*ArbolBB → Bool;
        [OP10] min:ArbolBB → Elem;
    VAR e,e1:Elem, ai,ad:ArbolBB;
    AXIOMAS
        [ABB9] inserta(e,nulo)=creanodo(nulo,e,nulo)
        [ABB10] inserta(e,creanodo(ai,e1,ad))=SI e<e1 ENT creanodo(inserta(e,ai),e1,ad)
                                              SI_NO SI e1<e ENT creanodo(ai,e1,inserta(e,ad))
                                                    SI_NO creanodo(ai,e1,ad) -- e=e1
        [ABB11] sustrae(e,nulo)=nulo
        [ABB12] sustrae(e,creanodo(ai,e1,ad))=
                    SI e<e1 ENT creanodo(sustrae(e,ai),e1,ad)
                    SI_NO SI e1<e ENT creanodo(ai,e1,sustrae(e,ad))
                          SI e=e1 y esnulo(ai) ENT ad
                          SI_NO SI e=e1 y esnulo(ad) ENT ai
                                SI_NO creanodo(ai,min(ad),sustrae(min(ad),ad))
                                -- e=e1 y no esnulo(ai) y no esnulo(ad)
```

```
[ABB13] miembro(e,nulo)=falso
[ABB14] miembro(e,creanodo(ai,e1,ad))=SI e<1 ENT miembro(e,ai)
                                      SI_NO SI e1<e ENT miembro(e,ad)
                                            SI_NO cierto          --e=e1
[AB15] min(nulo)=error
[AB16] min(creanodo(ai,e,ad))=SI esnulo(ai) ENT e
                              SI_NO min(ai)
FIN DE ArbolesDeBusquedaBinaria;
FIN DE ESQ ArbolesDeBusquedaBinariaEsq;
```

11.4 Paquete genérico ArbolesDeBusquedaBinaria

11.41 Interfaz

```
generic
    type Elem is private;
        with function "<" (Izq, Der : in Elem) return Boolean;

package ArbolesDeBusquedaBinaria is

    type ArbolBB is private;
    Arbol_Vacio: constant ArbolBB;
    procedure Vacia (El_Arbol: in out ArbolBB);
    function Izq (El_Arbol: in ArbolBB) return ArbolBB;
    function Der (El_Arbol: in ArbolBB) return ArbolBB;
    function Elemento (El_Arbol: in ArbolBB) return Elem;
    function Es_Vacio (El_Arbol: in ArbolBB) return Boolean;
    function Inserta (El_Nvo_Elemento: in Elem; En_El_Arbol: in ArbolBB)  return ArbolBB;
    function Sustrae (El_Elemento : in Elem;  Del_Arbol : in ArbolBB) return ArbolBB;
    function Es_Miembro (La_Clave : in Elem;   El_Arbol : in ArbolBB) return Boolean;
    function Min (El_Arbol : in ArbolBB) return Elem;

    Capacidad_Excedida : exception;
    Arbol_Es_Vacio : exception;

private
    type Nodo;
    type ArbolBB is access Nodo;
    Arbol_Vacio : constant ArbolBB := null;

end ArbolesDeBusquedaBinaria;
```

11.42 Cuerpo

```
package body ArbolesDeBusquedaBinaria is

    type Nodo is
        record
            El_Elemento : Elem;
            SubArbol_Izq : ArbolBB;
            SubArbol_Der : ArbolBB;
        end record;

procedure CreaNodo (El_Elemento: in Elem; El_Subarbol1: in ArbolBB;
                    El_Subarbol2: in ArbolBB; El_Arbol: in out ArbolBB) is
    begin
        El_Arbol := new Nodo'(El_Elemento => El_Elemento,  SubArbol_Izq => El_Subarbol1,
                              SubArbol_Der => El_Subarbol2);
    exception
            when storage_Error => raise Capacidad_Excedida;
    end CreaNodo;
```

11.4 Paquete genérico ArbolesDeBusquedaBinaria

11.41 Interfaz

```
generic
    type Elem is private;
        with function "<" (Izq, Der : in Elem) return Boolean;

package ArbolesDeBusquedaBinaria is

    type ArbolBB is private;
    Arbol_Vacio: constant ArbolBB;
    procedure Vacio (El_Arbol: in out ArbolBB);
    function Izq (El_Arbol: in ArbolBB) return ArbolBB;
    function Der (El_Arbol: in ArbolBB) return ArbolBB;
    function Elemento (El_Arbol: in ArbolBB) return Elem;
    function Es_Vacio (El_Arbol: in ArbolBB) return Boolean;
    function Inserta (El_Nvo_Elemento: in Elem; En_El_Arbol: in ArbolBB) return ArbolBB;
    function Sustrae (El_Elemento : in Elem;  Del_Arbol : in ArbolBB) return ArbolBB;
    function Es_Miembro (La_Clave : in Elem;   El_Arbol : in ArbolBB) return Boolean;
    function Min (El_Arbol : in ArbolBB) return Elem;

    Capacidad_Excedida : exception;
    Arbol_Es_Vacio : exception;

private
    type Nodo;
    type ArbolBB is access Nodo;
    Arbol_Vacio : constant ArbolBB := null;

end ArbolesDeBusquedaBinaria;
```

11.42 Cuerpo

```
package body ArbolesDeBusquedaBinaria is

    type Nodo is
        record
            El_Elemento : Elem;
            SubArbol_Izq : ArbolBB;
            SubArbol_Der : ArbolBB;
        end record;

procedure CreaNodo (El_Elemento: in Elem; El_Subarbol1: in ArbolBB;
                    El_Subarbol2: in ArbolBB; El_Arbol: in out ArbolBB) is
    begin
        El_Arbol := new Nodo'(El_Elemento => El_Elemento,  SubArbol_Izq => El_Subarbol1,
                              SubArbol_Der => El_Subarbol2);
    exception
            when storage_Error => raise Capacidad_Excedida;
    end CreaNodo;
```

```
procedure Vacio (El_Arbol : in out ArbolBB) is
begin
    El_Arbol := null;
end Vacio;

function Izq (El_Arbol : in ArbolBB) return ArbolBB is
begin
    return El_Arbol.SubArbol_Izq;
exception
    when Constraint_Error => raise Arbol_Es_Vacio;
end Izq;

function Der (El_Arbol : in ArbolBB) return ArbolBB is
begin
    return El_Arbol.SubArbol_Der;
exception
    when Constraint_Error => raise Arbol_Es_Vacio;
end Der;

function Elemento (El_Arbol : in ArbolBB) return Elem is
begin
    return El_Arbol.El_Elemento;
exception
    when Constraint_Error => raise Arbol_Es_Vacio;
end Elemento;

function Es_Vacio (El_Arbol : in ArbolBB) return Boolean is
begin
    return (El_Arbol = null);
end Es_Vacio;

function Inserta  (El_Nvo_Elemento: in Elem;  En_El_Arbol: in ArbolBB) return ArbolBB is
    Nodo_Temp : ArbolBB;
begin
    if En_El_Arbol = null then  CreaNodo (El_Nvo_Elemento, null, null, Nodo_Temp);
    elsif El_Nvo_Elemento = En_El_Arbol.El_Elemento then
                Nodo_Temp := En_El_Arbol;
    elsif El_Nvo_Elemento < En_El_Arbol.El_Elemento then
                CreaNodo (En_El_Arbol.El_Elemento,  Inserta(El_Nvo_Elemento,Izq(En_El_Arbol)),
                                                        Der(En_El_Arbol),  Nodo_Temp);
    else  CreaNodo (En_El_Arbol.El_Elemento, En_El_Arbol.SubArbol_Izq,
                        Inserta(El_Nvo_Elemento,En_El_Arbol.SubArbol_Der), Nodo_Temp );
    end if;
    return Nodo_Temp;
end Inserta;

function Sustrae (El_Elemento: in Elem; Del_Arbol:in ArbolBB) return ArbolBB is
    Arbol_Temp : ArbolBB;
begin
    if Del_Arbol = null then  Arbol_Temp :=  Arbol_Vacio;
    elsif El_Elemento = Del_Arbol.El_Elemento then
        if Del_Arbol.SubArbol_Der = null then  Arbol_Temp :=  Del_Arbol.SubArbol_Izq;
        elsif Del_Arbol.SubArbol_Izq = null then  Arbol_Temp :=  Del_Arbol.SubArbol_Der;
        else  CreaNodo(Min(Del_Arbol.SubArbol_Der),
                        Del_Arbol.SubArbol_Izq,
```

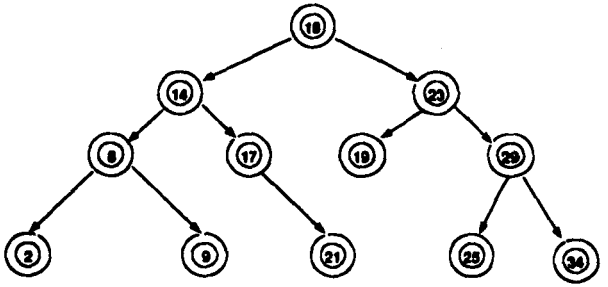
```
                    Sustrae(Min(Del_Arbol.SubArbol_Der),
                        Del_Arbol.SubArbol_Der), Arbol_Temp );
        end if;
    else if El_Elemento < Del_Arbol.El_Elemento then
            CreaNodo(Del_Arbol.El_Elemento,
                    Sustrae(El_Elemento, Del_Arbol.SubArbol_Izq),
                    Del_Arbol.SubArbol_Der, Arbol_Temp );
        else CreaNodo(Del_Arbol.El_Elemento,
                    Del_Arbol.SubArbol_Izq,
                    Sustrae(El_Elemento,Del_Arbol.SubArbol_Der), Arbol_Temp );
        end if;
    end if;
    return Arbol_Temp;
end Sustrae;

function Es_Miembro (La_Clave: in Elem; El_Arbol: in ArbolBB) return Boolean is
begin
    if El_Arbol = null then return FALSE;
    elsif El_Arbol.El_Elemento = La_Clave then return TRUE;
    elsif El_Arbol.El_Elemento < La_Clave then return
Es_Miembro(La_Clave,El_Arbol.SubArbol_Der);
    else return Es_Miembro(La_Clave,El_Arbol.SubArbol_Izq);
    end if;
end Es_Miembro;

function Min (El_Arbol : in ArbolBB) return Elem is
begin
    if not (El_Arbol = null) then
        if El_Arbol.SubArbol_Izq = null then return El_Arbol.El_Elemento;
        else return Min(El_Arbol.SubArbol_Izq);
        end if;
    end if;
exception
    when Constraint_Error => raise Arbol_Es_Vacio;
end Min;

end ArbolesDeBusquedaBinaria;
```

12. ÁRBOLES AVL

12.1 Definición

Un **árbol binario** es de altura balanceada o **AVL** (así llamado en honor de sus autores Adelson-Velskii y Landis) si está vacío o si ambos subárboles (izquierdo y derecho) son de altura balanceada, es decir, la diferencia existente entre las alturas de sus subárboles izquierdo y derecho es menor o igual a 1.

12.2 Diagrama de representación



Árbol AVL

12.3 Especificación algebraica ArbolesAVL

```
ESQUEMA ArbolesAVLEsq,
{
    PARAM ElementosOrd;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO;
        GENERO Elem;
        OPERACIONES
            _=_:Elem*Elem → Bool;
            _<_:Elem*Elem → Bool;
        VAR e1,e2,e3:Elem;
        AXIOMAS
            (e1=e1)=cierto
            (e1=e2)=(e2=e1)
            (e1=e2) y (e2=e3) ⇒ (e1=e3)=cierto
            (e1=e2) ⇒ (e1 <e2)=falso
            (e1<e2) y (e2<e3) ⇒ (e1<e3)=cierto
    FIN DE PARAM  ElementosOrd;
};
INSTANCIA ArbolesDeBusquedaBinariaEsq;
    RENOMBRA ArbolBB COMO ArbolAVL;
    CON ElementosOrd COMO ElementosOrd,
        Elem COMO Elem;
FIN DE INSTANCIA ArbolesDeBusquedaBinariaEsq;

ESPEC ArbolesAVL;
    IMPORTA TODO DE ElementosOrd;
    IMPORTA TODO DE ArbolesDeBusquedaBinariaEsq;
    EXPORTA ArbolAVL, nulo, miembro, ennulo, min DE ArbolesDeBusquedaBinaria;
            inserta,sustrae. DE ArbolesAVL;
    OPERACIONES
        [OP1] inserta:Elem*ArbolAVL → ArbolAVL
        [OP2] sustrae:Elem*ArbolAVL → ArbolAVL
        [OP3] rotaizq:ArbolAVL → ArbolAVL
        [OP4] rotader:ArbolAVL → ArbolAVL
        [OP5] balance_izq:ArbolAVL → ArbolAVL
        [OP6] balance_der:ArbolAVL → ArbolAVL
    VAR e,e1:Elem, ai,ad:ArbolAVL;
    AXIOMAS
        [A1] rotaizq(nulo)=nulo
        [A2] rotaizq(creanodo(ai,e,ad))=
                SI es_nulo(ad) ENT creanodo(ai,e,ad)
                SI_NO creanodo(creanodo(ai,e,izq(ad)),elem(ad),der(ad))
        [A3] rotader(nulo)=nulo
        [A4] rotader(creanodo(ai,e,ad))=
                SI es_nulo(ai) ENT creanodo(ai,e,ad)
                SI_NO creanodo(izq(ai),elem(ai),creanodo(der(ai),e,ad))
        [A5] inserta(e,nulo)=creanodo(nulo,e,nulo)
        [A6] inserta(e,creanodo(ai,e1,ad))=
                SI e<e1 ENT
            -- insercion en el subárbol izquierdo:
```

```
            SI altura (ai)<=altura (ad) ENT
        -- se inserta sin perder el balance
            creanodo(inserta(e,ai),el,ad)
          SI_NO    --altura(ai)>altura(ad) y la inserción causa desbalanceo.
        -- Hay dos casos:
            SI e<elem(ai) ENT rotader(creanodo(inserta(e,ai),e 1 ,ad))
            SI_NO
              SI elem(ai)<e ENT rotader(creanodo(rotaizq(inserta(e,ai)),el,ad))
              SI_NO          -- e=elem(ai),  no cambia nada
                creanodo(ai,el,ad)
        -- simétrico para la inserción en el subárbol derecho:
          SI_NO
            SI el< e ENT
              SI altura(ad)<=altura(ai) ENT
        -- se inserta sin perder el balance
                creanodo(ai,el ,inserta(e,ad))
              SI_NO
        -- altura(ad)>altura(ai) y la inserción causa desbalanceo. Hay dos casos:
                SI elem(ad)<e ENT rotaizq(creanodo(ai,e1,inserta(e,ad)))
                SI_NO
                  SI e<elem(ad) ENT rotaizq(creanodo(ai,e1,rotader(inserta(e,ad))))
                  SI_NO          -- e=elem(ad),  no cambia nada
                    creanodo(ai,e1,ad)
            SI_NO    -- e=e1,   no cambia nada
              creanodo(ai,e1,ad)
      [A7] balance izq(nulo)=nulo
      [A8] balance izq(creanodo(ai,e,ad))=
            SI altura(ad) - altura(ai)<=1 ENT
        -- no hay que hacer nada
              creanodo(ai,e, ad)
          SI_NO  -- altura(ad)-altura(ai)>=2,  hay dos casos:
              SI altura(izq(ad))>altura(der(ad)) ENT rotaizq(creanodo(ai,e,rotader(ad)))
              SI_NO rotaizq(creanodo(ai,e,ad)
      [A9] balanceder(nulo)=nulo
      [A10] balanceder(creanodo(ai,e,ad))= .
            SI altura(ai)-altura(ad)<=1 ENT
        -- no hay que hacer nada
              creanodo(ai,e,ad)
          SI_NO  -- altura(ai)-altura(ad)>=2
        -- Hay dos casos
              SI altura(der(ai))>altura(izq(ai)) ENT rotader(creanodo(rotaizq(ai),e,ad))
              SI_NO rotader(creanodo(ai,e,ad))
      [A11] sustrae(e,nulo)=nulo
      [A12] sustrae(e,creanodo(ai,el,ad))=
            SI e<e1 ENT balanceizq(creanodo(sustrae(e,ai),e1,ad))
            SI_NO  SI e1<e ENT balanceder(creanodo(ai,e1,sustrae(e.ad)))
                    SI_NO          --e=e1
                        SI esnulo(ai) ENT ad
                        SI_NO SI esnulo(ad) ENT ai
                                SI_NO     -- tiene dos hijos
                                    balanceizq(creanodo(sustrae(max(ai),ai),max(ai),ad))
FIN DE ArbolesAVL;
FIN DE ESQ ArbolesAVLEsq;
```

12.4 Paquete genérico ArbolesAVL

12.41 Interfaz

```
generic
   type Elem is private;
   with function "<" (Izq, Der : in Elem) return Boolean;

package ArbolesAVL is

   type ArbolAVL is private;
   Arbol_Vacio : constant ArbolAVL;
   procedure Vacio (El_Arbol : in out ArbolAVL);
   function Izq (El_Arbol : in ArbolAVL) return ArbolAVL;
   function Der (El_Arbol : in ArbolAVL) return ArbolAVL;
   function Elemento (El_Arbol : in ArbolAVL) return Elem;
   function Es_Vacio (El_Arbol : in ArbolAVL) return Boolean;
   function Inserta (El_Nvo_Elemento : in Elem;
                     En_El_Arbol : in ArbolAVL) return ArbolAVL;
   function Sustrae (El_Elemento : in Elem;
                     Del_Arbol : in ArbolAVL) return ArbolAVL;
   function Es_Miembro (La_Clave : in Elem;
                        El_Arbol : in ArbolAVL) return Boolean;
   function Min (El_Arbol : in ArbolAVL) return Elem;
   function Max (El_Arbol : in ArbolAVL) return Elem;

   Capacidad_Excedida : exception;
   Arbol_Es_Vacio : exception;

private
   type Nodo;
   type ArbolAVL is access Nodo;
   Arbol_Vacio : constant ArbolAVL := null;

end ArbolesAVL;
```

12.42 Cuerpo

```
package body ArbolesAVL is

   type Nodo is
      record
         El_Elemento : Elem;
         SubArbol_Izq : ArbolAVL;
         SubArbol_Der : ArbolAVL;
      end record;
```

```
-- Funciones de uso interno:

function CreaNodo (El_Elemento : in Elem;  El_Subarbol1 : in ArbolAVL;
                   El_Subarbol2 : in ArbolAVL) return ArbolAVL is
    Nodo_Temp : ArbolAVL;
begin
    Nodo_Temp := new Nodo'(El_Elemento => El_Elemento,
                           SubArbol_Izq => El_Subarbol1,
                           SubArbol_Der => El_Subarbol2);
    return Nodo_Temp;
exception
    when storage_Error => raise Capacidad_Excedida;
end CreaNodo;

function Altura (El_Arbol : in ArbolAVL) return Natural is
  function Max (Alt_Arb1, Alt_Arb2 : Natural) return Natural is
  begin
     if Alt_Arb1 > Alt_Arb2 then return Alt_Arb1;
     else return Alt_Arb2;
     end if;
  end Max;
begin
    if El_Arbol = null then return 0;
    else return (1 + Max( Altura(El_Arbol.SubArbol_Izq), Altura(El_Arbol.SubArbol_Der)));
    end if;
end Altura;

function Rota_Izq (El_Arbol : in ArbolAVL) return ArbolAVL is
    Arbol_Temp : ArbolAVL;
begin
    if El_Arbol = null then Arbol_Temp := Arbol_Vacio;
    elsif El_Arbol.SubArbol_Der = null then Arbol_Temp := El_Arbol;
    else Arbol_Temp := CreaNodo(Elemento(El_Arbol.SubArbol_Der),
                                CreaNodo(El_Arbol.El_Elemento,
                                         El_Arbol.SubArbol_Izq,
                                         Izq(El_Arbol.SubArbol_Der)),
                                Der(El_Arbol.SubArbol_Der));
    end if;
    return Arbol_Temp;
end Rota_Izq;

function Rota_Der (El_Arbol : in ArbolAVL) return ArbolAVL is
    Arbol_Temp : ArbolAVL;
begin
    if El_Arbol = null then Arbol_Temp := Arbol_Vacio;
    elsif El_Arbol.SubArbol_Izq = null then Arbol_Temp := El_Arbol;
    else Arbol_Temp := CreaNodo(Elemento(El_Arbol.SubArbol_Izq),
                                Izq(El_Arbol.SubArbol_Izq),
                                CreaNodo(El_Arbol.El_Elemento,
                                         Der(El_Arbol.SubArbol_Izq),
                                         El_Arbol.SubArbol_Der));
    end if;
    return Arbol_Temp;
end Rota_Der;
```

```
function Balance_Izq (El_Arbol : in ArbolAVL) return ArbolAVL is
      Arbol_Temp : ArbolAVL;
  begin
      if El_Arbol = null then Arbol_Temp := Arbol_Vacio;
      elsif altura(El_Arbol.SubArbol_Der) - altura(El_Arbol.SubArbol_Izq) <= 1 then
            Arbol_Temp := El_Arbol;
      elsif altura(Izq(El_Arbol.SubArbol_Der)) >  altura(Der(El_Arbol.SubArbol_Der)) then
            Arbol_Temp := Rota_Izq(CreaNodo(El_Arbol.El_Elemento,
                                            El_Arbol.SubArbol_Izq,
                                            Rota_Der(El_Arbol.SubArbol_Der)));
      else  Arbol_Temp := Rota_Izq(CreaNodo(El_Arbol.El_Elemento,
                                            El_Arbol.SubArbol_Izq,
                                            El_Arbol.SubArbol_Der));
      end if;
      return Arbol_Temp;
end Balance_Izq;

function Balance_Der (El_Arbol : in ArbolAVL) return ArbolAVL is
      Arbol_Temp : ArbolAVL;
begin
if El_Arbol = null then  Arbol_Temp := Arbol_Vacio;
elsif altura(El_Arbol.SubArbol_Izq) - altura(El_Arbol.SubArbol_Der) <= 1 then
            Arbol_Temp := El_Arbol;
elsif altura(Der(El_Arbol.SubArbol_Izq)) > altura(Izq(El_Arbol.SubArbol_Izq)) then
            Arbol_Temp := Rota_Der(CreaNodo(El_Arbol.El_Elemento,
                                            Rota_Izq(El_Arbol.SubArbol_Izq),
                                            El_Arbol.SubArbol_Der));
else  Arbol_Temp := Rota_Der(CreaNodo(El_Arbol.El_Elemento,
                                     El_Arbol.SubArbol_Izq,
                                     El_Arbol.SubArbol_Der));
end if;
      return Arbol_Temp;
end Balance_Der;
```

```
procedure Vacio (El_Arbol : in out ArbolAVL) is
begin
      El_Arbol := null;
end Vacio;

function Izq (El_Arbol : in ArbolAVL) return ArbolAVL is
begin
      return El_Arbol.SubArbol_Izq;
exception
      when Constraint_Error => raise Arbol_Es_Vacio;
end Izq;

function Der (El_Arbol : in ArbolAVL) return ArbolAVL is
begin
      return El_Arbol.SubArbol_Der;
exception
      when Constraint_Error => raise Arbol_Es_Vacio;
end Der;
```

```
function Elemento (El_Arbol : in ArbolAVL) return Elem is
begin
    return El_Arbol.El_Elemento;
exception
    when Constraint_Error => raise Arbol_Es_Vacio;
end Elemento;

function Es_Vacio (El_Arbol : in ArbolAVL) return Boolean is
begin
    return (El_Arbol = null);
end Es_Vacio;

function Inserta (El_Nvo_Elemento : in  Elem;  En_El_Arbol : in ArbolAVL) return ArbolAVL is
    Nuevo_Arbol : ArbolAVL;
begin
    if En_El_Arbol = null then  Nuevo_Arbol := CreaNodo (El_Nvo_Elemento, null, null);
    elsif El_Nvo_Elemento < En_El_Arbol.El_Elemento then
        if Altura(En_El_Arbol.SubArbol_Izq) <= Altura(En_El_Arbol.SubArbol_Der) then
            Nuevo_Arbol := CreaNodo(En_El_Arbol.El_Elemento,
                                    Inserta(El_Nvo_Elemento,En_El_Arbol.SubArbol_Izq),
                                    En_El_Arbol.SubArbol_Der);
            --Si altura(ai) > altura(ad) e inserción causa desbalanceo:
        elsif El_Nvo_Elemento < Elemento(En_El_Arbol.SubArbol_Izq) then
          Nuevo_Arbol:= Rota_Der(CreaNodo(En_El_Arbol.El_Elemento,
                                    Inserta(El_Nvo_Elemento,En_El_Arbol.SubArbol_Izq),
                                    En_El_Arbol.SubArbol_Der));
        elsif Elemento(En_El_Arbol.SubArbol_Izq) < El_Nvo_Elemento then
          Nuevo_Arbol := Rota_Der(CreaNodo(En_El_Arbol.El_Elemento,
                                    Rota_Izq(Inserta(El_Nvo_Elemento,
                                                En_El_Arbol.SubArbol_Izq)),
                                    En_El_Arbol.SubArbol_Der));
        else                -- Si e=elem(ai)
          Nuevo_Arbol := En_El_Arbol;          -- no cambia nada
        end if;
            -- Simétrico para la inserción en SubArbol derecho:
    elsif En_El_Arbol.El_Elemento < El_Nvo_Elemento  then
        if Altura(En_El_Arbol.SubArbol_Der) <= Altura(En_El_Arbol.SubArbol_Izq) then
          Nuevo_Arbol := CreaNodo(En_El_Arbol.El_Elemento,
                                  En_El_Arbol.SubArbol_Izq,
                                  Inserta(El_Nvo_Elemento,En_El_Arbol.SubArbol_Der));
            --Si altura(ad) > altura(ai) e inserción causa desbalanceo:
        elsif Elemento(En_El_Arbol.SubArbol_Der) <El_Nvo_Elemento then
          Nuevo_Arbol := Rota_Izq(CreaNodo (En_El_Arbol.El_Elemento,
                                    En_El_Arbol.SubArbol_Izq,
                                    Inserta(El_Nvo_Elemento,
                                        En_El_Arbol.SubArbol_Der)));
        elsif El_Nvo_Elemento < Elemento(En_El_Arbol.SubArbol_Der) then
    Nuevo_Arbol := Rota_Izq(CreaNodo (En_El_Arbol.El_Elemento,
                                  En_El_Arbol.SubArbol_Izq,
                                  Rota_Der(Inserta(El_Nvo_Elemento,
                                        En_El_Arbol.SubArbol_Der))));
        else Nuevo_Arbol := En_El_Arbol;          -- Si e=elem(ai)  no cambia nada
        end if;
    end if;
```

```
        return Nuevo_Arbol;
end Inserta;

function Sustrae (El_Elemento : in Elem;  Del_Arbol : in ArbolAVL) return ArbolAVL is
        Arbol_Temp : ArbolAVL;
begin
        if Del_Arbol = null then  Arbol_Temp :=  Arbol_Vacio;
        elsif El_Elemento < Del_Arbol.El_Elemento then
          Arbol_Temp := Balance_Izq(CreaNodo(Del_Arbol.El_Elemento,
                                             Sustrae(El_Elemento,Del_Arbol.SubArbol_Izq),
                                             Del_Arbol.SubArbol_Der));
        elsif Del_Arbol.El_Elemento < El_Elemento then
          Arbol_Temp := Balance_Der(CreaNodo (Del_Arbol.El_Elemento,
                                        Del_Arbol.SubArbol_Izq,
                                        Sustrae(El_Elemento,Del_Arbol.SubArbol_Der)));
              -- El_Elemento = Del_Arbol.El_Elemento:
        elsif Del_Arbol.SubArbol_Der = null then  Arbol_Temp :=  Del_Arbol.SubArbol_Izq;
        elsif Del_Arbol.SubArbol_Izq = null then   Arbol_Temp :=  Del_Arbol.SubArbol_Der;
        else                    -- tiene los dos hijos:
          Arbol_Temp := Balance_Izq(CreaNodo(Max(Del_Arbol.SubArbol_Izq),
                                             Sustrae(Max(Del_Arbol.SubArbol_Izq),
                                                     Del_Arbol.SubArbol_Izq),
                                             Del_Arbol.SubArbol_Der));
        end if;
        return Arbol_Temp;
end Sustrae;

function Es_Miembro (La_Clave : in Elem;  El_Arbol : in ArbolAVL) return Boolean is
begin
        if El_Arbol = null then  return FALSE;
        elsif El_Arbol.El_Elemento = La_Clave then  return TRUE;
        elsif El_Arbol.El_Elemento < La_Clave then  return
Es_Miembro(La_Clave,El_Arbol.SubArbol_Der);
        else  return Es_Miembro(La_Clave,El_Arbol.SubArbol_Izq);
        end if;
end Es_Miembro;

function Min (El_Arbol : in ArbolAVL) return Elem is
begin
        if not (El_Arbol = null) then  if El_Arbol.SubArbol_Izq = null then  return El_Arbol.El_Elemento;
                                       else  return Min(El_Arbol.SubArbol_Izq);
                                       end if;
        end if;
exception
        when Constraint_Error =>  raise Arbol_Es_Vacio;
end Min;
```

```
function Max (El_Arbol : in ArbolAVL) return Elem is
begin
    if not (El_Arbol = null) then
        if El_Arbol.SubArbol_Der = null then  return El_Arbol.El_Elemento;
        els  return Max(El_Arbol.SubArbol_Der);
        end if;
    end if;
exception
    when Constraint_Error => raise Arbol_Es_Vacio;
end Max;

end ArbolesAVL;
```

13. TABLAS HASH DE DICCIONARIOS

13.1 Definición

Un **diccionario** es un conjunto de parejas compuestas de **llaves** y **datos**.

Una **tabla** es una colección de un número arbitrario de elementos distintos (todos pertenecientes al mismo tipo), cada uno de los cuales se identifica mediante un **índice** distintivo (que, con frecuencia, es numérico) dentro de un **rango** determinado.

La función que transforma la llave de un diccionario en el índice de una tabla se llama función de **transformación** o función **hash**. Una función hash constituye un mapeo de los valores de las llaves al conjunto de los valores del rango. Su objetivo es el lograr una mayor uniformidad en la distribución de los datos guardados bajo las llaves, lo que aumentará la velocidad de búsqueda sin necesidad de tenerlos ordenados.

En los términos de la especificación abstracta de las tablas hash de diccionarios, se puede apreciar cómo la función hash relaciona las llaves y los rangos.

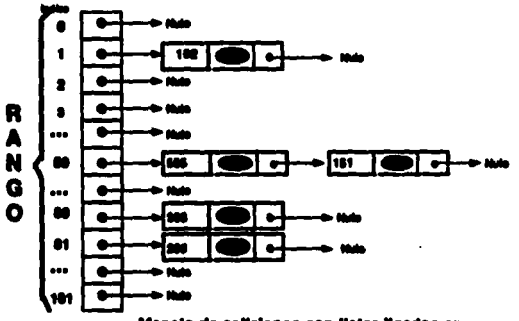
Lo ideal sería que la función hash seleccionada sólo diera lugar a un índice único para cada llave; desafortunadamente esto no es así, lo cual da lugar a los casos de **colisión**, es decir, situaciones en las que a diferentes llaves corresponde un mismo índice determinado mediante la aplicación de la función hash seleccionada.

Para resolver las colisiones en las tablas hash existen básicamente dos *métodos*: uno que asigna el índice a la primera llave que se presenta y para las siguientes señalará otro nuevo índice, que puede ser la siguiente localidad disponible o aquella localidad correspondiente a un índice de reasignación. El otro método, que emplea listas, agrega nodos ligados a un índice conforme se van presentando nuevas coincidencias del mismo índice.

En el mundo real existen muchos objetos que exteriorizan una estructura tipo diccionario básico, como los directorios telefónicos, las tablas de equivalencias, los diccionarios de términos, etcétera. Las tablas hash de diccionarios corresponden más a la implementación en sistemas de cómputo en donde podemos encontrar ejemplos de aplicaciones en el núcleo básico de muchos sistemas de bases de datos y en las tablas de símbolos de algunos compiladores.

13.2 Diagrama de representación

Suponiendo que el número de registros posibles sea 101, se establecerá como función hash h(x)=x MOD 101. De esta manera si se tiene un registro con llave 102, la aplicación de la función hash definida dará como resultado: 102 MOD 101 = 1, por lo que el registro será ubicado en la localidad 1.



Manejo de colisiones con listas ligadas en Tablas Hash de Diccionarios

13.3 Especificaciones algebraicas

Las implementaciones de los tipos de datos abstractos Diccionarios y Tablas no se implementaron porque ya existen como tipos de datos predefinidos bajo la forma de tipos de acceso (type access) y arreglos (type array).

Sin embargo, dada la estrecha relación existente entre los diccionarios, las tablas y las tablas hash de diccionarios, se presentarán las tres especificaciones mencionadas con el fin de facilitar la comprensión del tipo de datos abstracto Tablas hash de Diccionarios, como una composición de las especificaciones anteriores.

13.31 Diccionarios

```
ESQUEMA Diccionarios;
[
    PARAM Datos;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO;
        GENEROS Llave, Dato;
        OPERACION
            _=_:Llave*Llave →Bool;
        VAR  l,l1,l2:Llave;
        AXIOMAS
            (l=l)=cierto
            (l=l1)=(l1=l)
            (l=l1) y (l1=l2) ⇒ (l=l2)=cierto
    FIN DE PARAM Datos;
];

ESPEC Diccionarios;
    IMPORTA TODO DE Datos;
    EXPORTA TODO;
    GENERO  Dicc;
    OPERACIONES
        [OP1] vacio: → Dicc;
        [OP2] inserta:Llave*Dato*Dicc → Dicc;
        [OP3] elimina:Llave*Dicc → Dicc;
        [OP4] recupera:Llave*Dicc → Dato;
        [OP5] esvacio:Dicc → Bool;
        [OP6] miembro:Llave*Dicc → Bool;
    VAR l,l1:Llave,  d:Dato,  dic:Dicc;
    AXIOMAS
        [D1] elimina(l,vacio)=vacio
        [D2] elimina(l,inserta(l1,d,dicc))=SI  l=l1  ENT elimina(l,dic)
                                              SI_NO inserta(l1,d,elimina(l,dicc)
        [D3] recupera(l,vacio)=error
        [D4] max(creanodo(si,e,nd))=e
        [D5] recupera(l,inserta(l1,d,dicc))=SI l=l1 ENT d
                                              SI_NO  recupera(l,dicc)
        [D6] esvacio(inserta(l,d,dicc))=falso
        [D7] miembro(l,vacio)=falso
        [D8] miembro(l,inserta(l1,d,dic)))=SI l=l1 ENT cierto
                                              SI_NO miembro(l,dicc)
FIN DE Diccionarios;
FIN DE ESQ DiccionariosEsq;
```

13.32 Tablas

```
ESQUEMA TablasEsq;
[
    PARAM Elementos;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO;
        GENERO Elem;
    FIN DE PARAM Elementos;

    PARAM Rangos;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO;
        GENERO Rango;
        OPERACION
            _=_:Rango*Rango →Bool;
        VAR i,i1,i2:Llave;
        AXIOMAS
            (i=i)=cierto
            (i=i1)=(i1=i)
            (i=i1) y (i1=i2) ⇒ (i=i2)=cierto
    FIN DE PARAM Rangos;
];

ESPEC Tablas;
    IMPORTA TODO DE Elementos;
            TODO DE Rangos;
            TODO DE Booleanos;
    EXPORTA TODO;
    GENERO Tab;
    OPERACIONES
        [OP1] vacia: → Dicc;
        [OP2] asigna:Elem*Rango*Tab → Tab;
        [OP3] con:Rango*Tab → Elem;
        [OP4] asignado:Rango*Tab → Bool;
    VAR e:Elem, i,j:Rango, t:Tab;
    AXIOMAS
        [T1] con(i,vacia)=error
        [T2] con(i,asigna(e,j,t))=SI i=j ENT e SI_NO con(i,t)
        [T3] asignado(i,vacia)=falso
        [T4] asignado(i,asigna(e,j,t))= SI i=j ENT cierto SI_NO asignado(i,t)
FIN DE Tablas;
FIN DE ESQ TablasEsq;
```

13.33 Tablas Hash de Diccionarios

```
ESQUEMA TablasHashDeDiccionariosEsq;
[
    PARAM Datos;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO;
        GENEROS Llave, Dato;
        OPERACION
            _=_Llave*Llave → Bool
        VAR l,l1,l2:Llave;
        AXIOMAS
            [l1] (l=l)=cierto
            [l2] (l=l1)=(l1=l)
            [l3] (l=l1) y (l1=l2) ⇒ (l=l2)=cierto
    FIN DE PARAM Datos;

    PARAM Rangos;
        IMPORTA Bool DE Booleanos,
        EXPORTA TODO;
        GENERO Rango;
        OPERACION
            _=_:Rango*Rango → Bool
        VAR i,i1,i2:Rango;
        AXIOMAS
            [i1] (i=i)=cierto
            [i2] (i=i1)=(i1=i)
            [i3] (i=i1) y (i1=i2) ⇒ (i=i2)=cierto
    FIN DE PARAM Rangos;

    PARAM FuncionHash;
        IMPORTA Llave DE Datos;
                Rango DE Rango;
        EXPORTA TODO;
        OPERACION
            hash:Llave → Rango
    FIN DE PARAM FuncionHash;
];
INSTANCIA DiccionariosEsq
    CON Datos COMO Datos,
        Llave COMO Llave;
        Dato COMO Dato;
        _=_ COMO _=_;
FIN DE INSTANCIA DiccionariosEsq;

INSTANCIA TablasEsq;
    RENOMBRA Tab COMO TabHash;
    CON Elementos COMO Diccionarios,
        Elem COMO Dicc;
        Rangos COMO Rangos;
        Rango COMO Rango;
FIN DE INSTANCIA TablasEsq;
```

```
ESPEC TablasHashDeDiccionarios;
    IMPORTA TODO DE Datos;
            TODO DE Rangos;
            TODO DE FuncionHash;
            TODO DE Diccionarios;
            TODO DE Tablas;
            TODO DE Booleanos;
    EXPORTA TabHash, vacia DE Tablas;
            incluye, excluye, dato, existe DE TablasHashDeDiccionarios;
    OPERACIONES
        incluye:Llave*Dato*TabHash → TabHash;
        excluye:Llave*TabHash → TabHash;
        dato:Llave*TabHash → Dato;
        existe:LlaveTabHash → Bool;
    VAR  l:Llave, d:Dato, i:Rango, dic:Dicc, t:TabHash;
    AXIOMAS
        [THD1] incluye(l,d,vacia)=asigna(inserta(l,d,vacio),hash(l),vacia)
        [THD2] incluye(l,d,asigna(dic,i,t))= SI hash(l)=i ENT asigna(inserta(l,d,dic),i,t)
                                            SI_NO asigna(dic,i,incluye(l,d,t))
        [THD3] excluye(l,vacia)=vacia
        [THD4] excluye(l,asigna(dic,i,t))= SI hash(l)=i ENT asigna(elimina(l,dic),i,t)
                                            SI_NO asigna(dic,i,excluye(l,t))
        [THD5] dato(l,vacia)=error
        [THD6] dato(l,asigna(dic,i,t))= SI hash(l)=i ENT recupera(l,dic)
                                        SI_NO dato(l,t)
        [THD7] existe(l,vacia)=falso
        [THD8] existe(l,asigna(dic,i,t))= SI hash(l)=i ENT miembro(l,dic)
                                        SI_NO existe(l,t)
FIN DE TablasHashDeDiccionarios,
FIN DE ESQ TablasHashDiccionarioEsq.
```

13.4 Paquete genérico TablasHashDeDiccionarios

13.41 Interfaz

```
with Text_IO,IIO; use Text_IO;
generic
    type Llave is private;
    type Dato is private;
    Rango : in Positive;

    with function Hash (La_Llave : Llave; Rango:Positive) return Positive;
package TablasHashDeDiccionarios is

    type TabHash is limited private
    procedure TabHash_Vacia (La_TabHash: in out TabHash);
    procedure Incluye (La_Llave: in Llave; Y_El_Dato: in Dato;
                       En_La_TabHash: in out TabHash);
    procedure Excluye (La_Llave: in Llave; La_TabHash: in out TabHash);
    function FalDato (La_Llave: in Llave; La_TabHash: in TabHash) return Dato;
    function Existe (La_Llave: in Llave; La_TabHash: in TabHash) return
Boolean;

    Incluye_En_TabHash_Llena: exception;
    Llave_Ya_Asignada: exception;
    Llave_No_Asignada: exception;

private
    type Nodo;
    type Estructura is access Nodo;
    type TabHash is array (Positive range 1 .. Rango) of Estructura;

end TablasHashDeDiccionarios;
```

13.42 Cuerpo

```
package body TablasHashDeDiccionarios is

type Nodo is
    record
        La_Llave: Llave;
        El_Dato: Dato;
        Siguiente: Estructura;
    end record;
```

```
begin
    Nodo_Ant := null;
    La_Cubeta := (Hash(La_Llave)mod Rango) + 1;
    Nodo_Actual := En_La_TabHash(La_Cubeta);
    while Nodo_Actual /= null loop
     if Nodo_Actual.La_Llave = La_Llave then  return;
     else
         Nodo_Ant :=  Nodo_Actual;
         Nodo_Actual := Nodo_Actual.Siguiente;
     end if;
    end loop;
end Busca;

procedure TabHash_Vacia (La_TabHash: in out TabHash) is
-- Hacemos que cada cubeta apunte a nulo.
begin
    La_TabHash := TabHash'(others => null);
end TabHash_Vacia;

procedure Incluye (La_Llave: in Llave; Y_El_Dato: in Dato; En_La_TabHash: in out TabHash) is
    La_Cubeta: Positive;
    Nodo_Ant: Estructura;
    Nodo_Actual: Estructura;
begin
    Busca(La_Llave, En_La_TabHash, La_Cubeta, Nodo_Ant, Nodo_Actual);
    if Nodo_Actual /= null then  raise Llave_Ya_Asignada;
    else  En_La_TabHash(La_Cubeta) := new Nodo'(La_Llave  => La_Llave,
                                                El_Dato   => Y_El_Dato,
                                                Siguiente => En_La_TabHash(La_Cubeta));
    end if;
exception
    when Storage_Error => raise Incluye_en_TabHash_Llena;
end Incluye;

procedure Excluye (La_Llave: in Llave; La_TabHash: in out TabHash) is
    La_Cubeta: Positive;
    Nodo_Ant: Estructura;
    Nodo_Actual: Estructura;
begin
    Busca(La_Llave, La_TabHash, La_Cubeta, Nodo_Ant, Nodo_Actual);
    if Nodo_Ant = null then  La_TabHash(La_Cubeta) := Nodo_Actual.Siguiente;
    else  Nodo_Ant.Siguiente := Nodo_Actual.Siguiente;
    end if;
exception
    when Constraint_Error => raise Llave_No_Asignada;
end Excluye;
```

```
function FnDato (La_Llave: in Llave; La_TabHash: in TabHash) return Dato is
    La_Cubeta: Positive;
    Nodo_Ant: Estructura;
    Nodo_Actual: Estructura;
begin
    return Nodo_Actual.El_Dato;
exception
    when Constraint_Error => raise Llave_No_Asignada;
end FnDato;

function Existe (La_Llave: in Llave; La_TabHash: in TabHash) return Boolean is
    La_Cubeta: Positive;
    Nodo_Ant: Estructura;
    Nodo_Actual: Estructura;
begin
    Busca(La_Llave, La_TabHash, La_Cubeta, Nodo_Ant, Nodo_Actual);
    return (Nodo_Actual /= null);
end Existe;

end TablasHashDeDiccionarios;
```

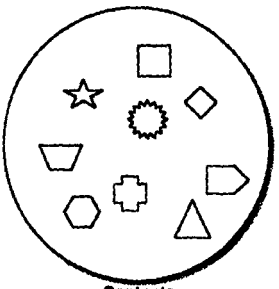
14. CONJUNTOS

14.1 Definición

Un **conjunto** es una colección no ordenada de elementos únicos obtenidos de una clase de objetos llamado Universo. La cardinalidad de un conjunto expresa cuántos elementos existen en el conjunto. Si no hay elementos en él, se le llama conjunto vacío. A los elementos que pertenecen a un conjunto se les considera miembros de ese conjunto.

Los conjuntos desempeñan un papel muy importante en la definición precisa de muchos conceptos en el mundo de las matemáticas y en el desarrollo de sistemas de software.

14.2 Diagrama de representación



14.3 Especificación algebraica

```
ESQUEMA ConjuntosEsq;
{
    PARAM Elementos;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO;
        GENERO Elem.
        OPERACION
            _=_:Elem*Elem → Bool;
    VAR e,el,e2:Elem.
    AXIOMAS
        [E1] (e=e)=cierto
        [E2] (e=el)=(el=e)
        [E3] (e=el) y (el=e2) ⇒ (e=e2)=cierto
    FIN DE PARAM Elementos;
};

ESPEC Conjuntos;
    IMPORTA TODO DE Elementos;
            TODO DE Booleanos;
            TODO DE Naturales;
    EXPORTA TODO;
    GENERO Conj;
    OPERACIONES
        [OP1] vacio: → Conj;
        [OP2] inserta:Elem*Conj → Conj;
        [OP3] elimina:Elem*Conj → Conj.
        [OP4] miembro:Elem*Conj → Bool;
        [OP5] esvacio:Conj → Bool;
        [OP6] union:Conj*Conj → Conj;
        [OP7] intersec:Conj*Conj → Conj;
        [OP8] dif:ConjConj → Conj;
        [OP9] card:Conj → Nat;
    VAR c,cl :Conj. e,el :Elem;
    AXIOMAS
        [C1] elimina(e,vacio)=vacio
        [C2] elimina(e,inserta(el,c))=SI e=el ENT elimina(e,c) SI_NO inserta(el,elimina(e,c))
             [C3] miembro(e,vacio)=falso
        [C4] miembro(e,inserta(el,c))=SI e=el ENT cierto SI_NO miembro(e,c)
        [C5] esvacio(vacio)=cierto
        [C6] esvacio(inserta(e,c))=falso
        [C7] union(c,vacio)=c
        [C8] union(c,inserta(e,c 1 ))=inserta(e,union(c,cl))
        [C9] intersec(c,vacio)=vacio
        [C10] intersec(c,inserta(e,cl))=SI miembro(e,c) ENT inserta(e,intersec(c,cl))
                                        SI_NO intersec(c,cl)
        [C8] dif(c,vacio)=c
        [C12] dif(c,inserta(e,cl))=SI miembro(e,c) ENT elimina(e,dif(c,cl)) SI_NO dif(c,cl)
        [C13] card(vacio)=0
        [C 14] card(inserta(e,c))=card(elimina(e,c))+ 1
FIN DE Conjuntos;
FIN DE ESQ ConjuntosEsq;
```

14.4 Paquete genérico Conjuntos

14.41 Interfaz

```
generic
    type Elem is private;

package CONJUNTOS is
    type Conj is limited private;
    Arbol_Vacio: constant Conj;
    procedure Vacio (El_Conjunto: in out Conj);
    function Inserta (El_Elemento: in Elem; En_El_Conjunto: in Conj) return Conj;
    function Elimina (El_Elemento: in Elem; Del_Conjunto: in Conj) return Conj;
    function Es_Miembro (La_Clave: in Elem; El_Conjunto: in Conj) return Boolean;
    function Es_Vacio (El_Conjunto: in Conj) return Boolean;
    function Union (El_Conjunto1: in Conj; El_Conjunto2 : in Conj) return Conj;
    function Interseс (El_Conjunto1: in Conj; El_Conjunto2: in Conj) return Conj;
    function Dif (El_Conjunto1: in Conj; El_Conjunto2: in Conj) return Conj;
    function Card (El_Conjunto: in Conj) return Natural;

    -- Declaraciones para el iterador pasivo
    generic
        with procedure Proceso (El_Elemento: in  Elem);
    procedure Iterar (Sobre_el_Conj: in Conj);

    Capacidad_Excedida: exception;
    Arbol_Es_Vacio: exception;

private
    type Nodo;
    type Conj is access Nodo;
    Arbol_Vacio: constant Conj := null;

end CONJUNTOS;
```

14.42 Cuerpo

```
package body CONJUNTOS is

    type Nodo is
        record
            El_Elemento: Elem;
            Siguiente: Conj;
        end record;

procedure Vacio (El_Conjunto: in out Conj) is
begin
    El_Conjunto := null;
end Vacio;
```

```
function Inserta (El_Elemento: in Elem; En_El_Conjunto: in Conj) return Conj is
    Nvo_Conj : Conj;
begin
    if es_miembro(El_Elemento,En_El_Conjunto) then  Nvo_Conj := En_El_Conjunto;
    else
        Nvo_Conj := new Nodo'(El_Elemento => El_Elemento,  Siguiente => En_El_Conjunto);
    end if;
    return Nvo_Conj;
exception
    when storage_Error =>  raise Capacidad_Excedida;
end Inserta;

function Elimina (El_Elemento: in Elem; Del_Conjunto: in Conj) return Conj is
    Conj_Temp : Conj;
begin
    if Del_Conjunto = null then  Conj_Temp :=  Arbol_Vacio;
    elsif Del_Conjunto.El_Elemento = El_Elemento then
        Conj_Temp := Elimina(El_Elemento,Del_Conjunto.Siguiente);
    else
        Conj_Temp := Inserta(Del_Conjunto.El_Elemento,
                             (Elimina(El_Elemento,Del_Conjunto.Siguiente)));
    end if;
    return Conj_Temp;
end Elimina;

function Es_Miembro (La_Clave: in Elem; El_Conjunto : in Conj) return Boolean is
begin
    if El_Conjunto = null then  return FALSE;
    elsif El_Conjunto.El_Elemento = La_Clave then  return TRUE;
    else  return Es_Miembro(La_Clave,El_Conjunto.Siguiente);
    end if;
end Es_Miembro;

function Es_Vacio (El_Conjunto: in Conj) return Boolean is
begin
    return (El_Conjunto = null);
end Es_Vacio;

function Card (El_Conjunto: in Conj) return Natural is
begin
    if El_Conjunto = null then  return 0;
    else return (1 + Card(Elimina(El_Conjunto.El_Elemento, El_Conjunto.Siguiente)));
    end if;
end Card;

function Union (El_Conjunto1: in Conj; El_Conjunto2: in Conj) return Conj is
Nvo_Conj : Conj;
begin
    if El_Conjunto2 = null then  Nvo_Conj := El_Conjunto1;
    else Nvo_Conj :=
inserta(El_Conjunto2.El_Elemento,Union(El_Conjunto1,El_Conjunto2.Siguiente));
    end if;
    return Nvo_Conj;
end Union;
```

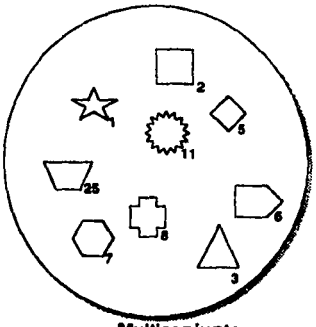
```
function Intersec (El_Conjunto1: in Conj; El_Conjunto2: in Conj) return Conj is
    Nvo_Conj : Conj;
begin
    if El_Conjunto2 = null then  Nvo_Conj := null;
    elsif Es_Miembro(El_Conjunto2.El_Elemento,El_Conjunto1) then
        Nvo_Conj := Inserta(El_Conjunto2.El_Elemento,
                            Intersec(El_Conjunto1, El_Conjunto2.Siguiente));
    else  Nvo_Conj := Intersec(El_Conjunto1,El_Conjunto2.Siguiente);
    end if;
    return Nvo_Conj;
end Intersec;

function Dif (El_Conjunto1: in Conj; El_Conjunto2: in Conj) return Conj is
    Nvo_Conj : Conj;
begin
    if El_Conjunto2 = null then
        Nvo_Conj := El_Conjunto1;
    elsif Es_Miembro(El_Conjunto2.El_Elemento,El_Conjunto1) then
        Nvo_Conj := Elimina(El_Conjunto2.El_Elemento,
                            Dif(El_Conjunto1,El_Conjunto2.Siguiente));
    else  Nvo_Conj := Dif(El_Conjunto1,El_Conjunto2.Siguiente);
    end if;
    return Nvo_Conj;
end Dif;

procedure Iterar (Sobre_el_Conj: in Conj) is
    El_Iterador : Conj := Sobre_el_Conj;
begin
    while not (El_Iterador = null) loop
        Proceso(El_Iterador.El_Elemento);
        El_Iterador := El_Iterador.Siguiente;
    end loop;
end Iterar;

end CONJUNTOS;
```

15. MULTICONJUNTOS

15.1 Definición

Un multiconjunto es una colección de elementos no ordenados que pueden estar duplicados obtenidos de una clase de objetos llamado el Universo.

15.2 Diagrama de representación



Multiconjunto

15.3 Especificación algebraica

```
ESQUEMA MultiConjuntosEsq;
[
    PARAM Elementos;
        IMPORTA Bool DE Booleanos;
        EXPORTA TODO
        GENERO Elem;
        OPERACION
            Elem*Elem → Bool;
        VAR e,e1,e2:Elem;
        AXIOMAS
            [E1] (e=e)=cierto
            [E2] (e=e1)=(e1=e)
            [E3] (e=e1) y (e1=e2) → (e=e2)=cierto
    FIN DE PARAM Elementos;
];
INSTANCIA ConjuntosEsq;
    RENOMBRA Conj COMO MConj;
             elimina COMO elimina1;
             card COMO card1;
    -- se ocultan las operaciones elimina y card de conjuntos y se definen de nuevo;
    -- las demás operaciones no sufren cambios.
    CON Elementos COMO Elementos;
FIN DE INSTANCIA ConjuntosEsq;

ESPEC MultiConjuntos,
    IMPORTA TODO DE Elementos;
            TODO DE Booleanos;
            TODO DE Naturales;
            TODO DE Conjuntos;
    EXPORTA MConj,vacio,inserta,miembro,esvacio, union,intersec,dif DE Conjuntos;
            elimina,card DE MultiConjuntos;
    OPERACIONES
        elimina:Elem*MConj → MConj;
        card:MConj → Nat;
    VAR e,e1 :Elem , c,c1 :Conj;
    AXIOMAS
        [C15] elimina(e,vacio)=vacio
        [C16] elimina(e,inserta(e1,c))= SI e=e1 ENT c
                                        SI_NO inserta(e1,elimina(e,c))
        [C17] card(vacio)=0
        [C 18] card(inserta(e,c))=card(c)+ 1
FIN DE MultiConjuntos;
FIN DE ESQ MultiConjuntosEsq;
```

15.4 Paquete genérico Multiconjuntos

15.41 Interfaz

```
generic
   type Elem is private;

package MULTICONJUNTOS is
   type MConj is limited private;
   Arbol_Vacio: constant MConj;
   procedure Vacio (El_MConjunto: in out MConj);
   function Inserta (El_Elemento: in Elem; En_El_MConjunto: in MConj) return MConj;
   function Elimina (El_Elemento: in Elem; Del_MConjunto: in MConj) return MConj;
   function Es_Miembro (La_Clave: in Elem; El_MConjunto : in MConj) return Boolean;
   function Es_Vacio (El_MConjunto: in MConj) return Boolean;
   function Union (El_MConjunto1: in MConj; El_MConjunto2 : in MConj) return MConj;
   function Intersec (El_MConjunto1: in Mconj; El_MConjunto2 : in MConj) return MConj;
   function Dif (El_MConjunto1: in MConj; El_MConjunto2: in MConj) return MConj;
   function Card (El_MConjunto: in MConj) return Natural;

   -- Declaraciones para el iterador pasivo
   generic
         with procedure Proceso (El_Elemento: in  Elem);
   procedure Iterar (Sobre_el_MConj: in MConj);

   Capacidad_Excedida: exception;
   Arbol_Es_Vacio: exception;

private
   type Nodo;
   type MConj is access Nodo;
   Arbol_Vacio: constant MConj := null;

end MULTICONJUNTOS;
```

15.42 Cuerpo

```
package body MULTICONJUNTOS is

      type Nodo is
            record
                  El_Elemento: Elem;
                  Siguiente: MConj;
            end record;

procedure Vacio (El_MConjunto: in out MConj) is
begin
      El_MConjunto := null;
end Vacio;
```

```
function Inserta (El_Elemento: in Elem; En_El_MConjunto: in MConj) return MConj is
    Nvo_MConj: MConj;
begin
    Nvo_MConj := new Nodo'(El_Elemento => El_Elemento, Siguiente => En_El_MConjunto);
    return Nvo_MConj;
exception
    when storage_Error => raise Capacidad_Excedida;
end Inserta;

function Elimina (El_Elemento: in Elem; Del_MConjunto: in MConj) return MConj is
    MConj_Temp: MConj;
begin
    if Del_MConjunto = null then MConj_Temp := Arbol_Vacio;
    elsif Del_MConjunto.El_Elemento = El_Elemento then MConj_Temp := Del_MConjunto.Siguiente;
    else MConj_Temp := Inserta(Del_MConjunto.El_Elemento,
                               (Elimina(El_Elemento,Del_MConjunto.Siguiente)));
    end if;
    return MConj_Temp;
end Elimina;

function Es_Miembro (La_Clave: in Elem; El_MConjunto: in MConj) return Boolean is
begin
    if El_MConjunto = null then return FALSE;
    elsif El_MConjunto.El_Elemento = La_Clave then return TRUE;
    else return Es_Miembro(La_Clave,El_MConjunto.Siguiente);
    end if;
end Es_Miembro;

function Es_Vacio (El_MConjunto: in MConj) return Boolean is
begin
    return (El_MConjunto = null);
end Es_Vacio;

function Card (El_MConjunto: in MConj) return Natural is
begin
    if El_MConjunto = null then return 0;
    else return (1 + Card(El_MConjunto.Siguiente));
    end if;
end Card;

function Union (El_MConjunto1: in Mconj; El_MConjunto2: in MConj) return MConj is
    -- La multiplicidad de sus elementos quedará reflejada en el nuevo conjunto.
    -- Su unión incluye repeticiones.
    Nvo_MConj : MConj;
begin
    if El_MConjunto2 = null then Nvo_MConj := El_MConjunto1;
    else Nvo_MConj := inserta(El_MConjunto2.El_Elemento,
                              Union(El_MConjunto1,El_MConjunto2.Siguiente));
    end if;
    return Nvo_MConj;
end Union;
```

```
function Intersec (El_MConjunto1: in MConj; El_MConjunto2: in MConj) return MConj is
    -- Aquí la multiplicidad se reduce a la menor posible.
    Nvo_MConj : MConj;
begin
    if El_MConjunto2 = null  then Nvo_MConj := null;
    elsif Es_Miembro(El_MConjunto2.El_Elemento,El_MConjunto1) then
        Nvo_MConj := Inserta(El_MConjunto2.El_Elemento,
                             Interscc(Elimina(El_MConjunto2.El_Elemento, El_MConjunto1),
                                      El_MConjunto2.Siguiente));
    else  Nvo_MConj := Interscc(El_MConjunto1,El_MConjunto2.Siguiente);
    end if;
    return Nvo_MConj;
end Interscc;

function Dif (El_MConjunto1: in MConj; El_MConjunto2 : in MConj) return MConj is
    Nvo_MConj : MConj;
begin
    if El_MConjunto2 = null then Nvo_MConj := El_MConjunto1;
    else  Nvo_MConj := Elimina(El_MConjunto2.El_Elemento,
                               Dif(El_MConjunto1,El_MConjunto2.Siguiente));
    end if;
    return Nvo_MConj;
end Dif;

procedure Iterar (Sobre_el_MConj: in MConj) is
    El_Iterador : MConj := Sobre_el_MConj;
begin
    while not (El_Iterador = null) loop
        Proceso(El_Iterador.El_Elemento);
        El_Iterador := El_Iterador.Siguiente;
    end loop;
end Iterar;
```

end MULTICONJUNTOS;

ANEXO B: PROGRAMA DE VALIDACIÓN DEL PAQUETE GENÉRICO COLAS CON PRIORIDADES

```
     with ColasConPrioridad;
procedure PRCOLPRI is

-- Declaraciones de tipos:
type Colores is (Amber, Blanco, Cafe, Gris, Negro);
type Alumno is record
          Cuenta : natural range 0..10 := 0;
          Apellido : string(1..10) := "          ";
          Nombre : string(1..10) := "          ";
end record;

-- Declaraciones de variables:
salirProgr : boolean;
salirOperac: boolean;
elem_color : Colores;
elem_reg  : Alumno;
sel_cola : integer;
sel_oper : integer;

-- declaración de función requerida para las instanciaciones de paquete:
function "<" (Izquierdo : in Alumno;
              Derecho   : in Alumno) return Boolean;

-- instanciaciones de paquete:
package Color_Io is new  Text_Io.Enumeration_Io(Colores);
package ColasPrEnum is new  ColasConPrioridad( Elem => Colores, "<"  => "<" );
package ColaPrRegs  is new  ColasConPrioridad( Elem => Alumno, "<"  => "<" );

-- declaraciones de variables de instanciaciones de paquete:
ColaPr1 : ColasPrEnum.ColaPr;
ColaPr2 : ColaPrRegs.ColaPr;

-- implementación de subprogramas requeridos por el procedimiento genérico Iterar:

procedure Despliega_Colores (El_Elemento : in Colores) is
begin
     New_Line;
     Color_Io.Put(El_Elemento);
end Despliega_Colores;

procedure Despliega_Reg (El_Elemento : in Alumno) is
begin
     New_Line;
     Put(El_Elemento.Cuenta); put("  ");
     Put(El_Elemento.Apellido); put ("  ");
     Put(El_Elemento.Nombre);
end Despliega_Reg;
```

```
-- Instanciaciones del procedimiento genérico Iterar:
procedure Despliega_ColasPrColores is new ColasPrEnum.Iterar(Proceso => Despliega_Colores);
procedure Despliega_ColasPrRegs is new ColasPrRegs.Iterar(Proceso => Despliega_Reg);

-- Cuerpo de la función "<" declarada anticipadamente
function "<" (Izquierdo : in Alumno;
              Derecho   : in Alumno) return Boolean is
begin
 if Izquierdo.Apellido < Derecho.Apellido then
     return True;
 elsif Izquierdo.Apellido = Derecho.Apellido then
     return ( (Izquierdo.Nombre) < (Derecho.Nombre) );
     else
      return False;
 end if;
end "<";

procedure MENU is

begin
 sel_cola := 0;
 loop
     new_line; new_line;
     put("- - - VALIDACION DEL PAQUETE GENERICO: COLAS CON PRIORIDADES - - -");
     new_line;
     put("Cola con prioridades de colores          [1]");
     new_line;
     put("Cola con prioridades de registros        [2]");
     new_line;
     put("Salir del programa                       [3]");
     new_line; new_line;
     put("Que opcion deseas  [1 .. 3] ? ");
     get(sel_cola);
     if sel_cola = 3 then
       salirProgr := TRUE;
     elsif (sel_cola > 3) then
       new_line;
       put("- - - No existe esa opcion !!! ");
       put("- - - Teclea nuevamente. ");
     end if;
     exit when (sel_cola=1) or (sel_cola=2) or (sel_cola=3);
 end loop;

end MENU;
```

```
procedure SUBMENU is
begin
 salirOperac := False;
 while not salirOperac loop
     new_line; new_line;
     put("- - - OPERACIONES SOBRE COLAS CON PRIORIDADES- - -");
     new_line;
     put("Crear una cola con prioridades vacia                    [1]");
     new_line;
     put("Meter elemento a la cola con prioridades                [2]");
     new_line;
     put("Sacar el elemento con valor maximo de la cola con prioridades  [3]");
     new_line;
     put("Ver el elemento con valor maximo de la cola con prioridades    [4]");
     new_line;
     put("Confirmar si la cola con prioridades esta vacia         [5]");
     new_line;
     put("Desplegar elementos de la cola con prioridades          [6]");
     new_line;
     put("Regresar al menu de tipos de colas con prioridades      [7]");
     new_line; new_line;
     put("Cual operacion deseas [1..7] ? ");
     get(sel_oper);
     case sel_oper is
      when 1 =>          -- crear una cola con prioridades vacia
      case sel_cola is
           when 1 =>     ColaPrEnum.Vacia(ColaPr1);
           when 2 =>     ColaPrRegs.Vacia(ColaPr2);
           when others => null;
      end case;
      when 2 =>          -- meter elemento de acuerdo con su prioridad
       new_line;
       put("Ingresa elemento: ");
       case sel_cola is
            when 1 =>
             new_line;
             put("[Posibilidades:  Ambar, Blanco, Cafe, Gris, Negro.]: ");
             Color_Io.get(elem_color);
             ColaPrEnum.Mete(elem_color, ColaPr1);
            when 2 =>
             put("Se trata de registros de alumnos. "); new_line;
             put("Escribe numero de cuenta: ");
             get(elem_reg.Cuenta); new_line;
             put("Escribe apellido [10 caracteres, rellena con espacios]: ");
             Text_Io.get(elem_reg.Apellido);   -- new_line;
             put("Escribe nombre de pila [10 caracteres, rellena con espacios]: ");
             Text_Io.get(elem_reg.Nombre);     --new_line;
             ColaPrRegs.Mete(elem_reg, ColaPr2);
             new_line;
            when others =>
             null;
       end case;
```

```
when 3 =>              -- sacar el elemento con valor maximo de la cola con prioridades
 new_line; new_line;
 put("Retire el elemento con valor maximo. ");
 case sel_cola is
      when 1 =>
       ColasPrEnum.Saca_Max(ColaPr1);
      when 2 =>
       ColaPrRegs.Saca_Max(ColaPr2);
      when others =>
       null;
 end case;
when 4 =>              --Ver el elemento con valor maximo de la cola con prioridades
 new_line;
 new_line;
 put("El elemento con valor maximo en la cola es: ");
 case sel_cola is
      when 1 =>
       elem_color := ColasPrEnum.Maximo(ColaPr1);
       Color_Io.put(elem_color);
       new_line;
      when 2 =>
       elem_reg := ColaPrRegs.Maximo(ColaPr2);
       --put(elem_reg);
       despliega_reg(elem_reg);
       new_line;
      when others =>
       null;
 end case;
when 5 =>              -- Confirmar si la cola con prioridades está vacia
 case sel_cola is
      when 1 =>
       if ColasPrEnum.Es_Vacia(ColaPr1) then
        Put_Line(" CIERTO: La cola esta vacia. ");
        new_line;
       else
        Put_Line(" FALSO: La cola NO esta vacia. ");
       end if;
      when 2 =>
       if ColaPrRegs.Es_Vacia(ColaPr2) then
        Put_Line(" CIERTO:La cola esta vacia. ");
        new_line;
       else
        Put_Line(" FALSO:La cola NO esta vacia. ");
       end if;
      when others =>
       null;
 end case;
when 6 =>              -- Desplegar elementos de la cola con prioridades
 new_line;
 case sel_cola is
      when 1 =>
       if ColasPrEnum.Es_Vacia(ColaPr1) then
        Put_Line(" La cola esta vacia. ");
        new_line;
```

```
            else  put("En la cola tengo como elementos: ");  new_line;
                  Despliega_ColasPrColores(ColaPr1);  new_line;
            end if;
           when 2 =>
            if ColaPrRegs.Es_Vacia(ColaPr2) then
             Put_Line(" La cola esta vacia. ");   new_line;
            else  put("En la cola tengo como elementos: ");  new_line;
             Despliega_ColaPrRegs(ColaPr2);  new_line;
       end if;
           when others =>
             null;
       end case;
      when 7 =>              --Regresar al menú de tipos de colas con prioridades
       new_line;
       Put("Cambiamos de tipo de cola. ");
       SalirOperac := TRUE;
      when others =>
       new_line;
       put("- - - Error en la entrada  !!! ");
       put("- - - Teclea nuevamente. ");
 end case;
end loop;
 exception
      when ColaPrEnum.Capacidad_Agotada | ColaPrRegs.Capacidad_Agotada=>
       new_Line;
       put(" ERROR: Trataa de sacar de una cola vacia. ");
       new_Line;
       put("     Regresamos al menu de tipos de colas.");
end SUBMENU;

                        -- PROGRAMA PRINCIPAL
begin
salirProgr := FALSE;
 loop
      MENU;
      exit when salirProgr;
      SUBMENU;
 end loop;
 new_line;
 put("   GRACIAS POR TU ATENCION. ");
 new_line; new_line;
 put("!!!QUE TENGAS UN MAGNIFICO DIA!!! ");
 new_line;
 exception
       when Io_exceptions.Data_error =>
          put ("- - - Oprimiste tecla equivocada. ");  new_line;
          put ("- - - Salimos del programa. ");
       when ColaPrEnum.Capacidad_Excedida
              | ColaPrRegs.Capacidad_Excedida =>
          put("Rebasamos capacidad de memoria. Salimos del programa. ");
end PRCOLIPRI;
```

BIBLIOGRAFÍA

Capítulo 1

[BIGGE89a] Biggerstaff, Ted J. and Alan J. Perlis, "*Introduction*" in Software Reusability Volume I Concepts and Models, edited by Ted Biggerstaff and Alan Perlis, ACM Press, U.S.A., 1989.

[BIGGE89b] Biggerstaff, Ted J. and Charles Richter, "*Reusability Framework, Assessment, and Directions*", in Software Reusability Volume I Concepts and Models, edited by Ted Biggerstaff and Alan Perlis, ACM Press, U.S.A., 1989.

[BIGGE89c] Biggerstaff, Ted J. and Alan J. Perlis, "*Introduction*", in Software Reusability Volume II, Applications and Experience, edited by Ted Biggerstaff and Alan Perlis, ACM Press, U.S.A., 1989.

[FEATH89] Feather, Martin S., "*Reuse in the Context of a Transformation-Based Methodology*" in Software Reusability Volume I Concepts and Models, edited by Ted Biggerstaff and Alan Perlis, ACM Press, U.S.A., 1989.

[GARGA87] Gargaro, Anthony and Frank Pappas, "*Reusability Issues and Ada*" in IEEE Transactions on Computers, U.S.A., July 1987

[HOROW89] Horowitz, Ellis and John B. Munson, "*An expansive View of Reusable Software*" in Software Reusability Volume I Concepts and Models, edited by Ted Biggerstaff and Alan Perlis, ACM Press, U.S.A., 1989.

[MEYER88] Meyer, Bertrand, "*Object-oriented Software Construction*", Prentice Hall International (UK) Ltd, Great Britain, 1988.

[NEIGH89] Neighbors, James M. "*Draco: A Method for Engineering Modules*" in Software Reusability Volume I Concepts and Models, edited by Ted Biggerstaff and Alan Perlis, ACM Press, U.S.A., 1989.

[PRESS89] Pressman, Roger S., "*Ingeniería de Software*" - Un enfoque práctico, McGraw-Hill, México, 2a. edición, 1989.

[PRIET89] Prieto-Díaz, Rubén, "*Classification of Reusable Modules*", in Software Reusability Volume I Concepts and Models, edited by Ted Biggerstaff and Alan Perlis, ACM Press, U.S.A., 1989.

[PRIET93] Prieto-Díaz, Rubén, "*Status Report: Software Reusability*" IEEE, May 1993.

[SOMME87] Sommerville, Ian & Ron Morrison, "*Software Development with Ada*", Addison-Wesley Publishing Company, Great Britain, 1987.

[WEGNE89] Wegner, Peter, "*Capital-Intensive Software Technology*", in Software Reusability Volume I Concepts and Models, edited by Ted Biggerstaff and Alan Perlis, ACM Press, U.S.A., 1989

Capítulo 2

[GOLDS85] Goldsack, S.J., *"Ada for specification: possibilities and limitations"*, Cambridge University Press, Great Britain, 1985.

[GUEZZ91] Ghezzi, Carlo, Mehdi Jazayeri & Dino Mandrioli, *"Fundamentals of Software Engineering"*, Prentice-Hall, Inc., New Jersey, U.S.A., 1991.

[HOARE69] Hoare, C.A.R., *"An Axiomatic Basis for Computer Programming"*, in Communications of the ACM, Volume 12, Number 10, October 1969.

[MARTI86] Martin, Johannes, J., *"Data Types and Data Structures"*, Prentice Hall International, Great Britain, 1986.

[OKTAB88] Oktaba, Hanna, Laura Espitia, Guadalupe Ibargüengoitia y Carlos Velarde, *"Especificación Formal en el Diseño de Programas"*, en Comunicaciones Técnicas, Serie Azul Monografías, No. 107, IIMAS-UNAM, México, 1988.

[OKTAB90] Oktaba, Hanna, *"Componentes reusables de software y su especificación formal"*, en Memorias del Taller de Programación Avanzada: Métodos y Lenguajes, Centro de Investigación en Matemáticas, Guanajuato, México, 1990.

[VANHO89] Van Horebeek and Johan Lewi, *"Algebraic Specifications in Software Engineering, An Introduction"*, Springer-Verlag, Germany, 1989.

Capítulo 3

[BOOCH87] Booch, Grady, *"Software Engineering with Ada"*, The Benjamin/Cummings Publishing Company, Inc., Second edition, U.S.A., 1987.

[BOOCH91] Booch, Grady, *"Object Oriented Design with Applications"*, The Benjamin/Cummings Publishing Company, Inc., U.S.A., 1991.

[BUDD91] Budd, Timothy, *"An introduction to Object-Oriented Programming"*, Addison-Wesley Publishing Company, Inc., U.S.A., 1991.

[COHEN86] Cohen, Norman H., *"Ada as a Second Language"*, McGraw-Hill Book Company, U.S.A., 1986.

[DUSIN87] Dusink, E.M.., *"Reflections on Reusable Software and Software Components"* in Proceedings of the Ada-Europe International Conference, Stockholm May 1987, edited by Sven Tafvelin, Cambridge University Press, Great Britain, 1987.

[GOLDS85] Goldsack, Stephen J., *"Ada for Specification: Possibilities and limitations"*, Cambridge University Press, Great Britain, 1985.

[HABER83] Habermann, A. Nico and Dewayne E. Perry, *"Ada for Experienced Programmers"*, Addison-Wesley, U.S.A., 1983.

[JONES90] Jones, Gregory W., *"Software Engineering"*, John Wiley & Sons, U.S.A., 1990.

[LEDGA83] Ledgard, Henry, *"Ada. An Introduction"*, Springer-Verlag, 1983.

[MEYER88] Meyer, Bertrand, *"Object-oriented Software Construction"*, Prentice Hall International (UK) Ltd, Great Britain, 1988.

[OCH593] Ocha, Tom, *"Ada-9X: Ada Comes of Age"*, in Software Development, November 1993.

[PRESS89] Pressman, Roger S., *"Ingeniería de Software* - Un enfoque práctico", McGraw-Hill, México, 2a. edición, 1989.

[QUINT93] Quintanilla, Gloria, *"El impacto en la Ingeniería de Software de la programación Orientada a Objetos"*, en Soluciones Avanzadas, Abril Mayo 1993

[SAMME86] Sammet, Jean E., *"Why Ada is not just another Programming Language"*, in Communicatios of the ACM, Volume 29 Number 3, August 1986.

[SOMME87] Sommerville, Ian and Ron Morrison, *"Software Development with Ada"*, Addison-Wesley Publishing Publishers Ltd., Great Britain, 1987

[WICHM84] Wichmann, Brian A., *"Is Ada too big? A designer answers the critics."* in Communications of the ACM, February 1984, Volume 27 Number 2.

[WIENE84] Wiener, Richard & Richard Sincovec, *"Software Engineering with Modula-2 and Ada"*, John Wiley & Sons, USA, 1984.

[WIENER89] Wiener, Richard & Richard Sincovec, *"Programación en Ada"*, Noriega Editores, Editorial Limusa, México, 1989.

Capítulo 4

[BOOCH87a] Booch, Grady, *"Software Engineering with Ada"*, The Benjamin/Cummings Publishing Company, Inc., U.S.A., 1983, Second Edition 1987.

[BOOCH87b] Booch, Grady, *"Software Components with Ada. Structures, Tools, and Subsystems"*, The Benjamin/Cummings Publishing Company, Inc., U.S.A., 1987.

[COHEN86] Cohen, Norman H., *"Ada as a Second Language"*, McGraw-Hill Company, U.S.A., 1986.

[HARRI89] Harrison, Rachel, *"Abstract Data Types in Modula-2"*, John Willey & Sons, Great Britain, 1989.

[LEDGA83] Ledgard, Henry, *"Ada. An Introduction"*, Springer-Verlag, U.S.A., 1983.

[LRM83] Department of Defense of the United States, *Military Standard of the Ada Programming Language*, ANSI/MIL-STD-1815A-1983

[MERID88] Meridian AdaVantage, *"Compiler User's Guide for PC-DOS Self Targeted Configurations"*, Compiler Version 2.1, Meridian Software Systems, Inc., U.S.A., 1988.

[SOMME87] Sommerville, Ian and Ron Morrison, *"Software Development with Ada"*, Addison-Wesley Publishing Company, Great Britain, 1987.

[WIENE84] Wiener, Richard y Richard Sincovec, *"Software Engineering with Modula-2 and Ada"*, John Wiley & Sons, U.S.A., 1984.

[WIENE89] Wiener, Richard y Richard Sincovec, *"Programación en Ada"*, Editorial Limusa, S.A. de C.V., México, 1989.