Programación en Ada

Índice general

Capítulo 1

Programación en Ada

1.1 Prólogo

El objetivo de este libro es aprender a programar en el lenguaje Ada, desde sus características más sencillas hasta las más avanzadas. Ada es un lenguaje potente, pero no por ello es más complicado que Pascal, por poner un ejemplo.

Los prerrequisitos son: nociones generales de programación y experiencia en otro lenguaje. Aunque si aún no se sabe programar se puede complementar con la lectura de otro manual destinado a ello.

Hay dos maneras de leer este libro. Una es desde el principio hasta el final, siguiendo el orden establecido en el índice y en los encabezados de cada sección. Otra es utilizar los enlaces libremente para saltar a los temas de interés elegidos por el lector.

1.1.1 Acerca de este libro

Este manual que se está editando ahora en Wikilibros, deriva del libro escrito por José Alfonso Malo Romero, antiguo profesor de la Universidad de Alcalá, para sus clases. Esa versión ya estaba licenciada según la GFDL, la misma licencia que usamos en Wikilibros, y se puede encontrar en varios formatos en MicroAlcarria.

El estado actual de desarrollo es: , lo que indica que aún hay trabajo por hacer. Recuerda que esto es un wiki: estás invitado a colaborar en la escritura de este libro. Es fácil, lee la bienvenida a los nuevos autores.

Si no quieres o no puedes contribuir pero quieres hacer sugerencias, como qué capítulos te gustaría ver más desarrollados o qué tema echas de menos, puedes hacerlo en la página de discusión.

1.2 Índice de contenidos

Ada Byron, Condesa de Lovelace

   (g) Registros
      i. Registros discriminados
      ii. Registros variantes
   (h) Punteros a objetos
   (i) Punteros a subprogramas
   (j) Tipos derivados
   (k) Tipos etiquetados (orientación a objetos)
8. Subtipos
9. Objetos (variables y constantes)
10. Atributos
11. Expresiones
12. Operadores
13. Sentencias y estructuras de control
14. Subprogramas
15. Sobrecarga
16. Entrada/salida
17. Pragmas
18. Interfaz con otros lenguajes
19. Cláusulas de representación
20. Diseño y programación de sistemas grandes
   (a) Paquetes
      i. Cláusula use
      ii. Cláusula with
      iii. Paquete Standard
   (b) Declaraciones, ámbito, visibilidad y renombrado
      i. Declaraciones
      ii. Ámbito
      iii. Visibilidad
      iv. Renombrado
   (c) La biblioteca *Ada* (unidades y subunidades)
      i. Unidades de biblioteca
      ii. Unidades hijas
      iii. Subunidades
      iv. Compilación separada y dependiente
   (d) Tipos abstractos de datos
   (e) Tipos limitados
   (f) Unidades genéricas
21. Excepciones
   (a) Paquete Ada.Exceptions
22. Concurrencia
   (a) Tareas
   (b) Sincronización de tareas mediante puntos de entrada o citas (entry)
      i. Aceptación de citas (accept)
      ii. Selección de citas (select)
      iii. Llamadas a punto de entrada complejas
   (c) Tareas dinámicas: creación dinámica de tareas (tipos tareas)
   (d) Dependencia de tareas
   (e) Unidades protegidas
   (f) Ejemplos de tareas
23. Programación distribuida con GLADE
24. Novedades de Ada 2005
25. Unidades predefinidas
   (a) Paquete System
   (b) Paquete Ada.Strings.Fixed
   (c) Paquete Ada.Strings.Bounded
   (d) Paquete Ada.Strings.Unbounded
   (e) Paquete Ada.Text_IO
   (f) Paquete Ada.Text_IO.Editing
   (g) Paquete Ada.Float_Text_IO
   (h) Paquete Ada.Integer_Text_IO
   (i) Paquete Ada.Sequential_IO
   (j) Paquete Ada.Calendar
   (k) Paquete Ada.Numerics
   (l) Paquete Ada.Command_Line
   (m) Paquete Interfaces
      i. Paquete Interfaces.C
26. Recursos en la Web
27. Guía de estilo

- Todo el texto en una página, ideal para imprimir. También existe una versión en PDF en canalada.org, aunque la versión más reciente siempre se encuentra en Wikilibros. Desde canalada.org también podemos descargar un paquete con todos los ejemplos del manual listos para compilar.

1.3 Recursos de Ada en la Web

- Artículo de Ada en Wikipedia
- Wikibook de programación en Ada, en inglés
- Ada en dmoz.org, directorio de Internet, rama en español
- AdaPower.com

- Ada World
- Foro en español sobre Ada
- Ada-Spain, asociación para la difusión del lenguaje en España

Más enlaces en Recursos en la Web

1.4 Créditos y licencia

Los autores de «Programación en Ada» son:

- José Alfonso Malo Romero, por la versión 1.0.1 (2001 - 2002)
- Por la actual versión de es.wikibooks.org (2004-2005):
  - Manuel Gómez (Contribuciones)
  - Andrés Soliño (Contribuciones)
  - Varios autores de «Ada Programming», por ciertos capítulos que han sido traducidos de ese libro (sus historiales así lo indican).
- Alvaro López Ortega, por la versión inicial de Programación en Ada/GLADE (2001).

Si quieres colaborar, sigue los consejos de «Cómo colaborar». Cuando hayas contribuido a la escritura del libro, añade tu nombre a la lista de autores.

Capítulo 2

Programación en Ada/Introducción

La programación de computadores, esto es, la creación de software para ellos, es un proceso de escritura. Así como los escritores de libros y artículos de revistas, los programadores (escritores de programas) deben expresar sus ideas por medio de una lengua escrita. Sin embargo, a diferencia de los escritores de novelas, los programadores deben expresar sus textos en lenguajes de propósito especial, basados en las matemáticas, conocidos como lenguajes de programación.

Ada, es uno de entre muchos posibles lenguajes de programación. Fue diseñado con un claro propósito en mente: la calidad del producto. Entendiéndose por calidad, la confianza que los usuarios van a poder depositar en el programa.

Si bien es posible escribir cualquier programa en Ada, éste ha sido utilizado principalmente, en el desarrollo de **software de control, de tiempo real y de misión crítica**. Típicamente, estos sistemas son responsables de procesos industriales y militares, muy costosos, y en los cuales incluso vidas humanas dependen del buen funcionamiento del software. Es vital en tales sistemas, utilizar un lenguaje que como Ada, ayuda en la creación de software de alta calidad.

Por otro lado, Ada, como lenguaje que promueve las buenas prácticas en ingeniería del software, es muy usado en la enseñanza de la programación en muchas universidades de todo el mundo.

Veamos cuáles son las características más destacables del lenguaje.

2.1 Características principales

Legibilidad Los programas profesionales se leen muchas más veces de las que se escriben, por tanto, conviene evitar una notación que permita escribir el programa fácilmente, pero que sea difícil leerlo excepto, quizás, por el autor original y no mucho tiempo después de escribirlo.

Tipado fuerte Esto asegura que todo objeto tenga un conjunto de valores que esté claramente definido e impide la confusión entre conceptos lógicamente distintos. Como consecuencia, el compilador detecta más errores que en otros lenguajes.

Construcción de grandes programas Se necesitan mecanismos de encapsulado para compilar separadamente y para gestionar bibliotecas de cara a crear programas transportables y mantenibles de cualquier tamaño.

Manejo de excepciones Los programas reales raramente son totalmente correctos. Es necesario proporcionar medios para que el programa se pueda construir en capas y por partes, de tal forma que se puedan limitar las consecuencias de los errores que se presenten en cualquiera de las partes.

Abstracción de datos Se puede obtener mayor transportabilidad y mejor mantenimiento si se pueden separar los detalles de la representación de los datos y las especificaciones de las operaciones lógicas sobre los mismos.

Procesamiento paralelo Para muchas aplicaciones es importante que el programa se pueda implementar como una serie de actividades paralelas. Dotando al lenguaje de estos mecanismos, se evita tener que añadirlos por medio de llamadas al sistema operativo, con lo que se consigue mayor transportabilidad y fiabilidad.

Unidades genéricas En muchos casos, la lógica de parte de un programa es independiente de los tipos de los valores que estén siendo manipulados. Para ello, se necesita un mecanismo que permita la creación de piezas de programa similares a partir de un único original. Esto es especialmente útil para la creación de bibliotecas.

Capítulo 3

Programación en Ada/Historia

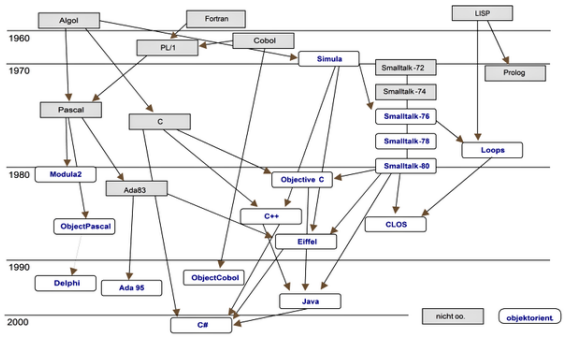


Gráfico de la historia de los lenguajes de programación. Las flechas indican relaciones de influyó a

La historia de Ada comienza en 1974 cuando el Ministerio de Defensa de los Estados Unidos llevó a cabo un estudio de los lenguajes de programación utilizados en sus proyectos y concluyó que *COBOL* era un estándar para procesado de datos y *FORTRAN* para cálculo científico. Sin embargo, la situación con respecto a los sistemas empotrados era diferente: el número de lenguajes en uso era enorme. Ante esta falta de estandarización que provocaba gastos inútiles, se propuso el uso de un único lenguaje para estos sistemas.

El primer paso del desarrollo fue la redacción en 1975 de un documento que perfilaba los requisitos que debía satisfacer el lenguaje. Después de varias modificaciones, en 1976 se produjo una versión de estos requisitos sobre la que se evaluaron varios lenguajes. El resultado fue que ninguno de los lenguajes existentes cumplía todos los requisitos. Así pues, el paso siguiente fue la aceptación de propuestas de diversos contratistas para el diseño de un nuevo lenguaje, de los que se eligieron cuatro de ellos. El siguiente paso fue el refinamiento de las propuestas elegidas y se revisaron las especificaciones para dar la versión definitiva conocida como *Steelman*.

La elección final del lenguaje se hizo en 1979 cuando se declaró vencedor el desarrollo de *CII Honeywell Bull*. Se decidió que se llamaría *Ada* en honor a *Augusta Ada Byron*, condesa de *Lovelace* (1815-1852), hija de *Lord Byron*, quien fue ayudante y patrocinadora de *Charles Babbage* trabajando en su máquina analítica mecánica, de hecho, está considerada por muchos como el primer programador de la historia.

Ada Lovelace (1838)

En 1983 se publicó el primer estándar ISO de Ada, el conocido Manual de referencia de Ada o ARM.

La primera revisión del lenguaje vino en 1995, marcando las dos versiones históricas que existen hasta el momento: Ada 83 y Ada 95. La última revisión ha sido aprobada recientemente por ISO y popularmente recibe el nombre de Ada 2005.

3.1 Enlaces externos

Más sobre la historia de Ada en:

- Artículo de Wikipedia
- Steelman language requirements
- Steelman On-Line, incluye el artículo "Ada, C, C++, and Java vs. the Steelman"

Capítulo 4

Programación en Ada/Manual de referencia

A menudo abreviado ARM o RM, el **manual de referencia de Ada** (nombre completo: Ada Reference Manual, ISO/IEC 8652:1995(E)), es el estándar oficial que define el lenguaje. A diferencia de otros documentos oficiales de ISO, el manual de Ada se puede reproducir libremente, lo cual es una gran ventaja para su difusión.

En este libro encontrarás enlaces al manual de referencia en cada sección acabada, en relación con el tema tratado. El ARM es el lugar donde todas las preguntas sobre Ada tienen respuesta, aunque puede ser un texto un tanto árido por su carácter de estándar oficial. En cualquier caso es la fuente más completa y exacta de información.

Otros documentos de cabecera de Ada son los *Ada Rationale*, documentos que acompañan al estándar y explican las justificaciones en el diseño de cada revisión del lenguaje.

Enlace externos al Manual de Referencia y Rationale

- Ada 2005:
  - Reference Manual y Rationale
- Ada 95
  - Ada 95 Reference Manual
  - Ada 95 Rationale
- Ada 83:
  - Ada 83 Reference Manual
  - Ada 83 Rationale

Capítulo 5

Programación en Ada/Instalación

En esta sección explicaremos cómo instalar un compilador de Ada. Existen dos opciones gratuitas:

- GNAT, compilador libre y gratuito.
- Versión demo de ObjectAda.

5.1 GNAT

GNAT es el único compilador de Ada gratuito completamente funcional. De hecho es parte del GNU Compiler Collection y por tanto es software libre. Sin embargo, la empresa que lo mantiene AdaCore, ofrece contratos de mantenimiento de las últimas versiones, antes de hacerlas públicas.

GNAT es también el único compilador que implementa todos los anexos (capítulos opcionales) del estándar de Ada.

5.1.1 GNAT GPL Edition: Ada 2005

Esta es la versión pública más reciente de AdaCore. Ten en cuenta que esta versión sólo permite distribuir binarios bajo la licencia GPL (la misma que usa Linux). Como ventaja decir que esta versión tiene soporte de las nuevas características de Ada 2005. Esta es la versión recomendable para estudiantes y profesionales que no necesitan soporte y licencian bajo GPL.

Para profesionales que quieren soporte o vender software bajo su propia licencia, AdaCore vende la edición llamada GNAT Pro.

GNAT GPL Edition se puede descargar de libre.adacore.com.

5.1.2 GNAT 3.15p precompilado

Para los que quieren distribuir binarios bajo una licencia distinta a la GPL y no pueden permitirse comprar una licencia de GNAT Pro lo más recomendable es descargar el paquete precompilado de la versión 3.15p, la última publicada por AdaCore bajo licencia GMGPL. Existen versiones para Windows, GNU/Linux y Solaris.

Aunque la versión precompilada no se ha actualizado recientemente, aún es suficiente para la mayoría de usuarios. Esta versión ha pasado el Ada Conformity Assessment Test Suite.

El Libre Site también proporciona el IDE para GNAT: GPS.

Se recomienda descargar estos paquetes adicionalmente:

- Linux
  - ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/mirrors/gnu-ada/3.15p/gnat-3.15p-i686-pc-redhat71-gnu-bin.tar.gz
  - ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/mirrors/gnu-ada/3.15p/florist-3.15p-src.tgz
  - ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/mirrors/gnu-ada/3.15p/asis/asis-3.15p-src.tgz
  - http://web.archive.org/web/20070820033755/http://libre.adacore.com/gps/gps-2.1.0-academic-x86-linux.tgz
- Windows
  - ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/mirrors/gnu-ada/3.15p/winnt/gnat-3.15p-nt.exe
  - ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/mirrors/gnu-ada/3.15p/winnt/gnatwin-3.15p.exe
  - ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/mirrors/gnu-ada/3.15p/asis/asis-3.15p-src.tgz
  - http://libre.adacore.com/gps/gps-2.1.0-academic-x86-windows.exe
- OS/2
  - ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/mirrors/gnu-ada/3.15p/contrib/os2/gnat-3.15p-os2-bin-20021124.zip

5.1.3 Paquetes precompilados formando parte de distribuciones

Existen varias distribuciones que contienen una copia del compilador GNAT, pero como consejo deberías comprobar la vesión de GCC:

gcc --version gcc (GCC) 3.4.3 Copyright (C) 2004 Free Software Foundation, Inc.

Versiones que comiencen con *3.3.* tienen muchos problemas por lo que deberías actualizarla a una vesión *3.4.*.

Mac OS X

GNAT for Macintosh proporciona una versión de GNAT, integración con Xcode y varios bindings.

Linux

La mayoría de las versiones de Linux vienen con paquetes de GNAT como parte de la distribución de GCC. Estas versiones se pueden usar perfectamente en vez de la versión de Libre Site.

SuSE Todas las versiones tienen un compilador GNAT incluído. La versión SuSE 9.2 y superiores contienen también paquetes de ASIS, Florist y GLADE.

Se necesitan estos paquetes:

gnat gnat-runtime

Debian GNU/Linux En Debian, GNAT se puede instalar con esta orden:

aptitude install gnat

Otros paquetes interesantes para el desarrollo en Ada bajo Debian 3.1 Sarge son:

- gnat - The GNU Ada 95 compiler
- ada-mode - Ada mode for GNU Emacs and XEmacs
- ada-reference-manual - The standard describing the Ada 95 language
- adabrowse - HTML generator for Ada 95 library unit specifications
- adacgi - Ada CGI interface
- asis-programs - Ada Semantic Interface Specification (ASIS) example programs
- gch - Ada quality & style checker
- gnade-dev - Develoment files for the GNat Ada Database Environment
- gnat-3.3 - The GNU Ada compiler
- gnat-3.4 - The GNU Ada compiler
- gnat-doc - Documentation for the GNU Ada compiler
- gnat-gdb - Ada-aware version of GDB
- gnat-glade - Distributed Systems Annex for GNAT (GNU Ada 95 compiler)
- gnat-gps - The GNAT Programming System - advanced IDE for C and Ada
- libadabindx-dev - Ada binding to the X Window System and *tif
- libadasockets0-dev - bindings for socket services in Ada
- libasis-3.15p-1-dev - Ada Semantic Interface Specification (ASIS) headers and li braries
- libaunit-dev - AUnit, a unit testing framework for Ada
- libaws-dev - Ada Web Server development files
- libflorist-3.15p-1-dev - POSIX.5 Ada interface to operating system services
- libgnomeada-2.4 - Ada binding for the Gnome Library
- libgtkada-2.4 - Ada binding for the GTK library
- libopentoken-dev - OpenToken lexical analysis library for Ada
- libtexttools-dev - Ada and C++ library for writing console applications
- libxmlada1-dev - XML/Ada, a full XML suite for Ada programmers
- libasis-3.14p-1 - Ada Semantic Interface Specification (ASIS) runtime library
- libcharles0-dev - Data structure library for Ada95 modelled on the C++ STL

Gentoo GNU/Linux Bajo Gentoo la instalación de GNAT es muy sencilla gracias al emerge:

emerge gnat

Mandrake En Mandrake, GNAT se puede instalar con esta orden:

urpmi gnat

Windows

Tanto MinGW para Windows como Cygwin contienen un paquete de GNAT.

MinGW MinGW - Minimalist GNU for Windows.

Esta lista puede ayudarte a instalarlo:

1. Instalar MinGW-3.1.0-1.exe.
   (a) extraer binutils-2.15.91-20040904-1.tar.gz.
   (b) extraer mingw-runtime-3.5.tar.gz.
   (c) extraer gcc-core-3.4.2-20040916-1.tar.gz.
   (d) extraer gcc-ada-3.4.2-20040916-1.tar.gz.
   (e) extraer gcc-g++−3.4.2-20040916-1.tar.gz (opcional).
   (f) extraer gcc-g77-3.4.2-20040916-1.tar.gz (opcional).
   (g) extraer gcc-java-3.4.2-20040916-1.tar.gz (opcional).
   (h) extraer gcc-objc-3.4.2-20040916-1.tar.gz (opcional).
   (i) extraer w32api-3.1.tar.gz.
2. Instalar mingw32-make-3.80.0-3.exe (opcional).
3. Instalar gdb-5.2.1-1.exe (opcional).
4. Instalar MSYS-1.0.10.exe (opcional).
5. Instalar msysDTK-1.0.1.exe (opcional).
   (a) extraer msys-automake-1.8.2.tar.bz2 (opcional).
   (b) extraer msys-autoconf-2.59.tar.bz2 (opcional).
   (c) extraer msys-libtool-1.5.tar.bz2 (opcional).

Se recomienda usar D:\MinGW como directorio de instalación.

Es de notar que la versión para Windows del Libre Site también se basa en MinGW.

Cygwin Cygwin, el entorno GNU para Windows, también contiene una versión de GNAT, aunque más antigua que la de MinGW, y no soporta DLLs ni multi-tarea (al 11/2004).

MS-DOS

DJGPP es un porte a MS-DOS de una selección de herramientas GNU con extensiones de 32 bits, que está mantenido activamente. Incluye la colección completa de compiladores de GCC, lo cual ahora incluye Ada. Véase el sitio de DJGPP para instrucciones de instalación.

Los programas de DJGPP, a parte de en modo nativo MS-DOS, se pueden ejecutar en ventanas de DOS en Windows.

Solaris 8, 9, 10 sobre SPARC y x86

Descarga GCC de blastwave.org. Es la *suite* completa con el compilador de GNAT.

5.1.4 Compilarlo uno mismo

Si quieres las últimas características, incluyendo el soporte experimental de Ada 2005, tendrás que compilar tú mismo el GNAT.

Instrucciones:

- Linux o Cygwin: véase ada.krischik.com.
- Windows: *pendiente*

5.2 Aonix ObjectAda

Aonix ObjectAda es un compilador comercial que proporciona una versión gratuita de evaluación, con limitaciones en el número y tamaño de archivos y en el número de controles que se pueden colocar en una ventana.

El IDE de ObjectAda es muy similar al de MS Visual C++ y al de Delphi e incluye un *GUI builder*.

5.2.1 Enlaces

- Aonix ObjectAda
- Versión demo

Capítulo 6

Programación en Ada/Hola Mundo

Un ejemplo común de la sintaxis de un lenguaje es el programa Hola mundo. He aquí una implementación en Ada con la intención de ser un primer contacto.

with Ada.Text_IO; **procedure** Hola_Mundo **is begin** Ada.Text_IO.Put_Line("¡Hola, mundo!"); **end** Hola_Mundo;

Por ahora puede ser suficiente con aprender a compilar y enlazar un programa escrito en Ada, pero si tienes curiosidad aquí va una explicación del programa.

La cláusula with establece una dependencia con el paquete Ada.Text_IO y hace disponible toda la funcionalidad relacionada con la Entrada/Salida de textos.

Después se define un procedimiento como programa principal. Nótese que en Ada no tiene que tener un nombre especial como main, simplemente ha de estar fuera de cualquier paquete. Al compilar se indica cuál es el programa principal.

Ada.Text_IO.Put_Line llama al procedimiento Put_Line definido en el paquete Ada.Text_IO para imprimir la cadena "¡Hola, mundo!".

6.1 Abreviando

Si no deseamos tener sentencias largas, como Ada.Text_IO.Put_Line, se pueden usar dos métodos para abreviar:

1. Usar una cláusula use para hacer directamente visible todas las entidades definidas en Ada.Text_IO. No es recomendable abusar de esta cláusula por las razones explicadas aquí.
2. Usar una cláusula renames, para renombrar Ada.Text_IO con un nombre más corto.

-- *Con clásula use* **with** Ada.Text_IO; **use** Ada.Text_IO; **procedure** Hola_Mundo **is begin** Put_Line("¡Hola, mundo!"); **end** Hola_Mundo; -- *Con cláusula renames* **with** Ada.Text_IO; **use** Ada.Text_IO; **procedure** Hola_Mundo **is package** T_IO **renames** Ada.Text_IO; **begin** T_IO.Put_Line("¡Hola, mundo!"); **end** Hola_Mundo;

6.2 Compilación

Como ejemplo, con el compilador GNAT este programa se debe escribir en un archivo llamado hola_mundo.adb (el nombre del procedimiento que contiene, más .adb) y se compilaría así:

gnatmake hola_mundo.adb

Naturalmente si usas un entorno integrado de desarrollo la compilación será una opción de menú o un botón de la barra de herramientas.

El resultado es un archivo ejecutable llamado hola_mundo que imprime *¡Hola, mundo!* por su salida estándar (normalmente en una ventana en modo texto).

Capítulo 7

Programación en Ada/Elementos del lenguaje

7.1 Alfabeto

El alfabeto de *Ada* consta de:

- Letras mayúsculas: A, ..., Z y minúsculas: a, ..., z.
- Dígitos: 0, ..., 9.
- Caracteres especiales.

Es de destacar que en Ada 95 se admiten caracteres como 'Ñ', 'Ç' y vocales acentuadas ya que se permiten los 256 caracteres comprendidos en *ISO Latin-1*.

El alfabeto de minúsculas puede usarse en vez de o junto con el alfabeto de mayúsculas, pero se considera que los dos son idénticos (a excepción de las cadenas de caracteres y literales tipo carácter).

7.2 Componentes léxicos

Se pueden encontrar en Ada los siguientes componentes léxicos:

- Identificadores
- Literales numéricos
- Literales de tipo carácter
- Cadenas de caracteres
- Delimitadores
- Comentarios
- Palabras reservadas

Hacer constar, que el espacio no constituye nada más que un separador de elementos léxicos, pero es muy importante utilizarlos para una mayor legibilidad, tanto dentro de las sentencias, como elemento de sangrado para ayudar a diferenciar los bloques.

Ejemplo:

Temperatura_Sala := 25; -- *Temperatura que debe tener la sala.*

Esta línea contiene 5 elementos léxicos:

- El identificador Temperatura_Sala
- El delimitador compuesto :=
- El número 25
- El delimitador simple ;
- El comentario -- *Temperatura que debe tener la sala.*

7.2.1 Identificadores

Definición en *BNF*:

identificador ::= letra { [subrayado] letra | cifra } letra ::= A | ... | Z | a | ... | z cifra ::= 0 | ... | 9 subrayado ::= _

Aunque dentro de esta definición entrarían las palabras reservadas que tienen un significado propio en el lenguaje y, por tanto, no pueden ser utilizadas como identificadores.

Nota: en la versión *Ada 95* se incorporan los caracteres de *Latin-1*, con lo que se pueden escribir identificadores como Año o Diámetro.

No hay límite en el número de caracteres de un identificador, aunque todo identificador deberá caber en una única línea.

Como en cualquier lenguaje, es recomendable utilizar nombres significativos como Hora_Del_Dia y no H, carente de significado.

Problema: **¿Son identificadores Ada validos estas palabras? En caso negativo, ¿por qué razón? _Hora_Del_Día, Inicio_, Mañana, Hora_Del_Día, Jabalí, contador, 2a_vuelta, ALARMA, Access, Precio_en_$, alarma__general, HoraDelDía.**

Solución:

consejo

ejercicio

_Hora_Del_Día: no, porque comienza por guión bajo.
Inicio_: no, porque termina por guión bajo.
Mañana: sí.
Hora_Del_Día: sí.
Jabalí: sí.
contador: sí.
2a_vuelta: no, porque comienza por número.
ALARMA: sí.
Access: no, es una palabra reservada de Ada.
**Precio_en_$**: no, contiene un carácter ($) que no es letra, cifra ni guión bajo.
alarma__general: no, porque contiene dos guiones bajos seguidos.
HoraDelDía: sí.

7.2.2 Números

Los literales numéricos constan de:

- dígitos 0 .. 9
- el separador de decimales .
- el símbolo de exponenciación e o E
- el símbolo de negativo -
- el separador _

Como ejemplo, el número real 98,4 se puede representar como: 9.84E1, 98.4e0, 984.0e-1 ó 0.984E+2. No estaría permitido 984e-1.

Para representación de número enteros, por ejemplo 1.900, se puede utilizar 19E2, 190e+1 ó 1_900E+0. Sirviendo el carácter _ como mero separador para una mejor visualización.

Una última característica es la posibilidad de expresar un literal numérico en una base distinta de 10 encerrando el número entre caracteres #, precedido por la base: un número entre 2 y 16. Por ejemplo, 2#101# equivale a 101 en base binaria, es decir al número 5 en decimal. Otro ejemplo con exponente sería 16#B#E2 que es igual a $11 \times 16^2 = 2.816$ (nótese que es 16^2 y no 10^2 porque la base en este caso es 16).

7.2.3 Literales de tipo carácter

Contienen un único carácter, por ejemplo: A. Aquí sí se diferencian mayúsculas de minúsculas. Se delimitan por un apóstrofe.

Ejemplos:

'A' 'ñ' '%'

7.2.4 Cadenas de caracteres

Contienen uno o varios caracteres y se delimitan por el carácter de dobles comillas: ", por ejemplo: “ABC”. En este caso se diferencian mayúsculas de minúsculas.

7.2.5 Delimitadores

Los delimitadores pueden ser uno de los siguientes caracteres especiales:

& ' ( ) * + , - . / : ; < = >

O ser uno de los delimitadores compuestos por dos caracteres especiales:

=> .. ** := /= >= <= << >> <>

7.2.6 Comentarios

Los comentarios se utilizan para ayudar a comprender los programas y lo constituye toda parte de texto precedida de dos guiones (--) hasta el fin de línea. No existe la posibilidad de insertar otro elemento léxico en la misma línea a partir de los dos guiones, es decir, el resto de la línea se interpreta como comentario en su totalidad.

-- *Este comentario ocupa una línea completa.* Mis_Ahorros := Mis_Ahorros * 10.0; -- *Este está después de una sentencia.* Mis_Ahorros := Mis_Ahorros * -- *Este está entre medias de una sentencia que ocupa dos líneas.* 100_000_000.0;

7.2.7 Palabras reservadas

Como el resto de los elementos léxicos, las palabras reservadas de Ada son equivalentes tanto en mayúsculas como en minúsculas. El estilo más extendido es escribirlas completamente en minúsculas.

En Ada las palabras reservadas pueden tener un uso distinto dependiendo del contexto, los distintos usos de cada una se puede consultar en el capítulo Palabras reservadas.

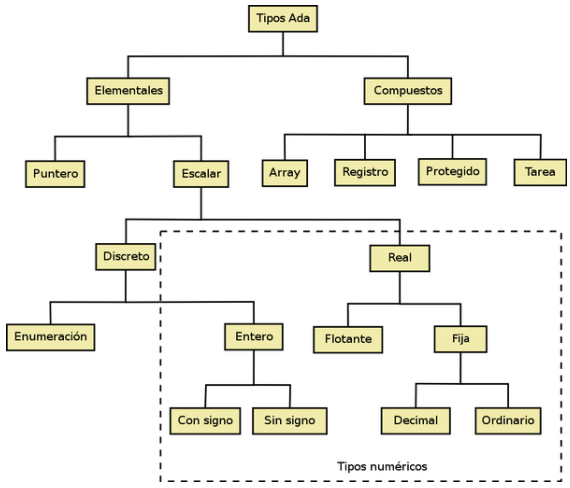
7.3 Manual de referencia de Ada

- Section 2: Lexical Elements

Capítulo 8

Programación en Ada/Tipos

8.1 Clasificación de tipos



Jerarquía de tipos en Ada

Los tipos de *Ada* se pueden clasificar en:

- Escalar:
  - Discreto:
    - Enteros: Integer, Natural, Positive.
    - Enumeraciones: Boolean, Character.
  - Real:
    - Coma flotante.
    - Coma fija.
- Compuesto:
  - Vector: Arrays, Strings.
  - Registros: **record**.
- Puntero: **access**.
  - Punteros a objetos
  - Punteros a subprogramas
- Privados: **private**.
- Tareas: **task**.

8.2 Declaración de tipos

Para definir un tipo no estándar, se puede emplear el siguiente esquema:

declaración_tipo ::= **type** identificador **is** definición_tipo ;

Sirvan como ejemplos las siguientes definiciones de tipos:

-- *Escalares discretos no estándar:* -- **type** TIndice **is range** 1..50; -- *Escalares reales no estándar:* -- **type** TPeso **is digits** 10; -- *Coma flotante con precisión de 10 cifras.* **type** TMasa **is delta** 0.01 **range** −12.0 .. 12.0; -- *Coma fija 0.01 precis.* -- *Enumeración:* -- **type** TColor **is** (ROJO, VERDE, AZUL); -- *Vectores:* -- **type** TMatriz **is array** (1..10, 1..10) **of** Float; **type** TVector5 **is array** (TIndice **range** 5..10) **of** Float; -- *Registros:* -- **type** TVálvula **is record** Nombre: String(1..20); Abierta: Boolean; VelocidadLíquido: Float **range** 0.0 .. 30.0; **end record**; -- *Punteros:* -- **type** PEntero **is access** Integer; -- *Arrays irrestringidos. Tienen un número indefinido de* -- *elementos. Es necesario especificar los límites al declarar* -- *variables de ese tipo.* -- **declare type** TVectorIndef **is array** (Integer **range** <>) **of** Float; V: TVectorIndef (1..4); **begin** V(1) := 10.28; V := (1.2, 1.5, 1.8, 1.3); V := (1 => 1.2, 2 => 1.7, **others** => 0.0); **end**;

8.3 Algunos atributos aplicables a tipos

Los atributos son operaciones predefinidas que se pueden aplicar a tipos, objetos y otras entidades. Por ejemplo estos son algunos atributos aplicables a tipos:

- **Last**: Integer'Last es el máximo valor que puede tomar la variable de tipo Integer. También es el último valor de un tipo enumerado o del índice de un vector.
- **First**: Integer'First es el mínimo valor que puede tomar la variable de tipo Integer. También es el primer valor de un tipo enumerado o del índice de un vector.

- **Range**: Vector'Range indica el rango que ocupa la variable Vector, es decir, equivale a Vector'First..Vector'Last. En el caso de más de una dimensión, el valor Matriz'Range(1) indica el rango de la primera dimensión.

- **Succ**: TColor'Succ(ROJO) indica el siguiente valor a ROJO que toma el tipo TColor, si no existe, se eleva la excepción Constraint_Error.

- **Pred**: TDía'Pred(VIERNES) indica el anterior valor a VIERNES que toma el tipo TDía, si no existe, se eleva la excepción Constraint_Error.

- **Pos**: El atributo Pos indica la posición ocupada por un determinado valor en un tipo enumeración. Por ejemplo: TColor'Pos(ROJO). La primera posición se considera 0.

- **Val**: El atributo Val indica el valor que ocupa una determinada posición en un tipo enumeración. Por ejemplo: COLOR'Val(1).

- Para identificar unívocamente un valor de un tipo enumeración se emplea TColor' (ROJO) y TIndicador'(ROJO) para distinguir el valor ROJO del tipo TColor o TIndicador.

8.4 Subtipos

Los subtipos definen un subconjunto de los valores de un tipo determinado, pero no son un tipo distinto de su tipo base.

8.5 Superar una ambigüedad

En el supuesto caso de que se quiera superar una ambigüedad en el tipo de una variable (debería evitarse) en un determinado instante, se puede optar por convertirlo (no recomendable) o cualificarlo:

- Convertir el tipo, por ejemplo: Integer(I) convierte el número I a entero.

- Cualificar el tipo, por ejemplo: Integer'(I) interpreta I como entero.

En ambos casos, el resultado es I como entero.

La cualificación sirve para cuando un literal no se sabe a qué tipo pertenece y se le indica de ese modo. Por ejemplo:

... type Color is (Rojo, Azul, Naranja); type Fruta is (Naranja, Pera, Melon); procedure Put (Una_Fruta : Fruta); procedure Put (Un_Color : Color); ... begin Put (Color'(Naranja)); -- Llama a procedure Put (Un_Color : Color) Put (Fruta'(Naranja)); -- Llama a procedure Put(Una_Fruta : Fruta) Put (Naranja); -- Error de sintaxis, la llamada es ambigua Put (Rojo); -- Ok, llama a procedure Put (Un_Color : Color) end;

8.6 Tipos avanzados

Hay otros tipos más avanzados en Ada. Puedes optar por leer estos tipos ahora o por continuar el libro por la siguiente lección Objetos (variables y constantes).

- Registros discriminados
- Registros variantes
- Punteros a objetos
- Punteros a subprogramas
- Tipos derivados
- Tipos etiquetados

8.7 Manual de referencia de Ada

- 3.2 Types and Subtypes
- 3.2.1 Type Declarations

Capítulo 9

Programación en Ada/Tipos/Enteros

9.1 Tipos enteros con signo

Un tipo entero con signo se define declarando un rango, por ejemplo:

type Índice **is range** 1 .. 50;

Los extremos del rango se pueden consultar con los atributos 'First y el 'Last del tipo.

Cuando se asigna un valor a una variable de este tipo, se chequea en tiempo de ejecución si pertenece al rango. En caso negativo se levanta la excepción Constraint_Error.

El compilador puede definir el tamaño más eficiente para el tipo a no ser que se defina una cláusula de representación.

type Índice **is range** 1 .. 50; **for** Índice'Size **use** 8;

9.1.1 Ejemplo

El siguiente ejemplo, tomado del wikilibro en inglés sobre Ada, define un nuevo rango de -5 a 10 e imprime el rango completo.

-- File: range_1.adb (view) (download) **with** Ada.Text_IO; **procedure** Range_1 **is type** Range_Type **is range** -5 .. 10; **package** T_IO **renames** Ada.Text_IO; **package** I_IO **is new** Ada.Text_IO.Integer_IO (Range_Type); **begin for** A **in** Range_Type **loop** I_IO.Put (Item => A, Width => 3, Base => 10); **if** A < Range_Type'Last **then** T_IO.Put (","); **else** T_IO.New_Line; **end if**; **end loop**; **end** Range_1;

9.1.2 Tipos enteros con signo predefinidos

En el paquete Standard se predefinen varios tipos enteros con signo, como Integer (y sus subtipos Natural y Positive), Short_Integer, Long_Integer y posiblemente otros (dependiendo del compilador). Estos tipos tienen los tamaños más adecuados para la arquitectura del computador. Si no tienes razones para definir nuevos tipos enteros, considera usar estos tipos o un subtipo de ellos.

Esta sería una posible representación gráfica de los subtipos de Integer:

9.2 Enteros modulares

Los enteros modulares no tienen signo y son cíclicos, es decir no se produce desbordamiento (*overflow*) sino *wrap-around*. Para estos tipos están predefinidos los operadores lógicos (and, or, xor) para realizar operaciones a nivel de bits.

Se definen así:

type *Nombre* **is mod** *Módulo*;

Donde el 'First es 0 y 'Last es Módulo - 1. Como los tipos modulares son cíclicos $last + 1 \equiv first$ y $first - 1 \equiv last$.

Se puede definir un subtipo de un tipo modular:

type Byte **is mod** 256; **subtype** Medio_Byte **is** Byte **range** 0 .. 127;

Pero, cuidado porque el módulo de Medio_Byte sigue siendo 256.

9.3 Manual de referencia de Ada

- 3.5.4 Integer Types

Capítulo 10

Programación en Ada/Tipos/Enumeraciones

Un tipo **enumeración** es una lista definida de los posibles valores que puede adoptar un objeto de ese tipo. La declaración:

type Color_Primario **is** (Rojo, Verde, Azul);

establece que a un objeto del tipo Color_Primario se le puede asignar cualquiera de los tres valores indicados, y ningún otro valor. Como los tipos numéricos, donde por ejemplo 1 es un literal entero, Rojo, Verde y Azul son los llamados literales del tipo.

Las enumeraciones son uno de los tipos discretos, los otros son los tipos enteros.

10.1 Operadores y atributos predefinidos

Aparte del operador de igualdad ("="), los tipos enumeración tienen predefinidos todos los operadores de orden: "<", "<=", "=", "/=", ">=", ">"; donde la relación de orden viene dada implicitamente por la secuencia de los literales: cada literal tiene una posición, empezando por 0 para el primero, que se incrementa en uno para cada sucesor. Esta posición se puede obtener mediante el atributo 'Pos; el inverso es 'Val, que devuelve el valor correspondiente a una posición. En nuestro ejemplo:

Color_Primario'Pos (Rojo) = 0 Color_Primario'Val (0) = Rojo

Los literales pueden sobrecargarse, es decir, puede haber otro tipo con los mismos literales.

type Luz_de_Tráfico **is** (Rojo, Ambar, Verde);

Normalmente se puede determinar por el contexto a qué tipo pertenece el literal Rojo. Cuando es imposible, el programador ha de usar una expresión de desambiguación: Tipo'(Valor).

10.2 Literales carácter

Una característica bastante única de Ada es la posibilidad de usar literales caracter como literales de una enumeración.

type ABC **is** ('A', 'B', 'C');

El literal 'A' no tiene mucho que ver con el literal 'A' del tipo predefinido Character (o Wide_Character).

Todo tipo que tiene al menos un literal carácter es un tipo carácter. Para todo tipo carácter, existen literales de cadena y el operador de concatenación predefinido "&".

type Mi_Caracter **is** (No_Caracter, 'a', Literal, 'z'); **type** Mi_String **is array** (Positive **range** <>) **of** Mi_Caracter; S: Mi_String := “aa” & Literal & “za” & 'z'; T: Mi_String := ('a', 'a', Literal, 'z', 'a', 'z');

En este ejemplo, S y T contienen el mismo valor.

El tipo predefinido Character se define de este modo en el paquete Standard.

10.3 Tipo boolean

El tipo predefinido *boolean* es una enumeración definida en el paquete Standard de este modo:

type Boolean **is** (False, True);

Por esta razón el tipo boolean tiene todos los atributos definidos para cualquier otra enumeración.

10.4 Subtipos de enumeración

Se puede usar un rango para definir un subtipo sobre una enumeración:

subtype Letra_Mayúscula **is** Character **range** 'A' .. 'Z'; **subtype** Color_Frío **is** Color_Primario **range** Verde .. Azul;

10.5 Manual de referencia de Ada

- 3.5.1 Enumeration Types

Capítulo 11

Programación en Ada/Tipos/Coma flotante

Para definir un tipo de **coma flotante** es suficiente con definir cuantos dígitos se necesitan de este modo:

digits *Digitos*

Si se desea también se puede definir el rango mínimo:

digits *Digitos* **range** *Primero .. Último*

Esta capacidad es uno de los grandes beneficios de Ada sobre la mayoría de los lenguajes de programación en este respecto. Otros lenguajes, sólo proporcionan un tipo *float* y otro *long float*, y lo que la mayoría de los programadores hacen es:

- elegir *float* si no están interesados en la precisión
- de lo contrario, eligen *long float*, puesto que es lo mejor que pueden obtener.

En cualquiera de los dos casos, el programador no sabe cual es la precisión que obtiene.

En Ada, uno especifica la precisión necesitada y el compilador elige el tipo de coma flotante que cumple *al menos* esa precisión. De este modo el requisito se cumple. Además, si la máquina tiene más de dos tipos de coma flotante, el compilador puede hacer uso de todos ellos.

Por supuesto, el programador también puede hacer uso de los tipos de coma flotante predefinidos que son *Float* y posiblemente (si el compilador lo implementa) *Short_Float*, *Short_Short_Float*, *Long_Float* y *Long_Long_Float*.

11.1 Manual de referencia de Ada

- 3.5.7 Floating Point Types

Capítulo 12

Programación en Ada/Tipos/Coma fija

12.1 Coma fija decimal

Es posible definir un tipo de coma fija decimal declarando el delta (el error absoluto) y la cantidad de dígitos en base decimal necesitados (incluyendo la parte real):

delta *Delta* **digits** *Digitos*

Delta ha de ser una potencia de 10, si no, el tipo no será de coma fija decimal.

También podemos definir el rango mínimo necesitado:

delta *Delta* **digits** *Digitos* **range** *Primero .. Último*

Ejemplo:

type T_Precio_en_Euros **is delta** 0.01 **digits** 15;

12.2 Coma fija ordinaria

Para un tipo de coma fija ordinaria (binaria) se especifica simplemente el delta con un rango.

delta *Delta* **range** *Primero .. Último*

Ejemplo:

type T_Medida **is delta** 0.125 **range** 0.0 .. 255.0;

12.3 Manual de referencia Ada

- 3.5.9 Fixed Point Types

Capítulo 13

Programación en Ada/Tipos/Arrays

Un array es una colección de elementos a los que se puede acceder por su índice. En Ada todo tipo definitivo (aquel del que se conoce su tamaño) se permite como elemento y cualquier tipo discreto, (es decir, enteros con signo y modulares, o enumeraciones) puede usarse como índice.

13.1 Declaración de arrays

Los arrays de Ada son de alto nivel, comparados, por ejemplo, con los de C/C++. Esto se traduce en varias posibilidades sintácticas que se presentan a continuación.

13.1.1 Sintáxis básica

La declaración básica de un array es la siguiente:

array (Tipo_Índice) **of** Tipo_Elemento

Este array consiste en un elemento de tipo *Tipo_Elemento* por cada posible valor de Tipo_Índice. Por ejemplo, si quisieramos contar las ocurrencias de cada letra en un texto nos definiríamos un array de este tipo:

type Contador_Caracteres **is array** (Character) **of** Natural;

> *Nota: usamos Natural como tipo de elemento puesto que los valores negativos de Integer no tienen sentido en una cuenta. Es conveniente usar el subtipo entero más adecuado en cada caso, puesto que así nos beneficiamos de la comprobación de rango y podemos descubrir errores fácilmente.*

13.1.2 Con subrango conocido

A menudo no necesitamos un array con todos los valores posibles del tipo del índice. En este caso definimos un subtipo del tipo índice con el rango necesitado.

subtype Subtipo_Índice **is** Tipo_Índice **range** Primero ... Último; **array** (Subtipo_Índice) **of** Tipo_Elemento;

Para lo que hay una forma más abreviada si no deseamos definir el subtipo con nombre, se puede hacer anónimamente:

array (Tipo_Índice **range** Primero ... Último) **of** Tipo_Elemento;

Puesto que Primero y Último son expresiones del tipo Tipo_Índice, una forma más simple es la siguiente:

array (Primero ... Último) **of** Tipo_Elemento

Ten en cuenta que si First y Last son literales numéricos, esto implica que el tipo índice base es el Integer.

Si en el ejemplo anterior, sólo deseásemos contar letras mayúsculas desechando otros caracteres, podríamos definir el tipo array de este modo:

type Contador_Caracteres **is array** (Character **range** 'A' .. 'Z') **of** Natural;

13.1.3 Con un subrango desconocido

A menudo el rango necesitado no se conoce hasta tiempo de ejecución o necesitamos objetos array de varias longitudes. En lenguajes de más bajo nivel como C necesitaríamos hacer uso de la memoria dinámica (del *heap*). Pero no es el caso de Ada, puesto que la caja <> nos permite declarar arrays de tamaño no restringido:

array (Tipo_Índice **range** <>) **of** Tipo_Elemento

Cuando declaramos objetos de este tipo, los extremos (*bounds*) del array deben conocerse, bien como resultado de una función o por una inicialización mediante un agregado. Desde su declaración hasta su finalización, el objeto no puede cambiar de tamaño.

13.1.4 Con elementos *aliased*

Los programadores de C/C++ dan por hecho que todo elemento de un array tiene una dirección propia en memoria (de hecho el nombre del array es un puntero sobre el que se puede operar).

En Ada, esto no es siempre así. Veamos este ejemplo:

type Día_De_Mes **is range** 1 .. 31; **type** Día_Con_Cita **is array** (Día_De_Mes) **of** Boolean; **pragma** *Pack*

(Día_Con_Cita);

Puesto que hemos empaquetado el array, el compilador usará el mínimo espacio de almacenamiento posible. En la mayoría de los casos esto implica que los 8 valores booleanos cabrán en un byte.

Pero este no es el único caso en el que el compilador de Ada puede empaquetar un array puesto que tiene libertad en los casos en que sea más optimo.

Si queremos acceder con un puntero a cada elemento tenemos que expresarlo explícitamente.

type Día_De_Mes **is range** 1 .. 31; **type** Día_Con_Cita **is array** (Día_De_Mes) **of aliased** Boolean;

13.2 Uso de arrays

Para acceder a los elementos de un array se usan el nombre del objeto array seguido del índice entre paréntesis.

Se puede acceder a una rodaja (*slice*) de un array usando (x .. y).

Vector_A (1 .. 3) := Vector_B (4 .. 6);

El operador "&" permite concatenar arrays:

Nombre_Completo := Nombre & ' ' & Apellidos;

Si se intenta acceder a un elemento más alla de los límites del array o se asigna a un array (completo o slice) un array de distinto tamaño se levanta la excepción Constraint_Error (a menos que los chequeos estén deshabilitados).

13.2.1 Ejemplo de uso

with Ada.Text_IO, Ada.Integer_Text_IO; **use** Ada.Text_IO, Ada.Integer_Text_IO; **procedure** Agenda **is type** Dia_De_Mes **is** range 1 .. 31; **type** Dia_Con_Cita **is** array (Dia_De_Mes) **of** Boolean; Citas_En_Mayo : Dia_Con_Cita := (**others** => False); -- *Se inicializa todo el mes a False* **begin** -- *Tengo citas los dias 3, del 8 al 16 (excepto el 14), y el último día del mes.* Citas_En_Mayo (3) := True; Citas_En_Mayo (8 .. 16) := (**others** => True); Citas_En_Mayo (14) := False; Citas_En_Mayo (Citas_En_Mayo'Last) := True; Put (“En mayo tienes citas los dias:"); **for** I **in** Citas_En_Mayo'Range **loop if** Citas_En_Mayo (I) **then** Put (Dia_De_Mes’Image (I)); **end if**; **end loop**; **end** Agenda;

13.3 Véase también

- Fundamentos de programación/Estructuras de datos/Arrays

13.4 Manual de referencia de Ada

- 3.6 Array Types

Capítulo 14

Programación en Ada/Tipos/Strings

En Ada los strings son un tipo especial de array que está implicitamente definido en el paquete Standard como:

type String **is array** (Positive **range** <>) **of** Character;

Que el rango se defina sobre el subtipo Positive implica que ningún string de Ada puede empezar en 0. Esta es una diferencia con los strings de C/C++, la otra es que un string de Ada no tiene por qué terminar en NUL (carácter de código 0), de hecho puede tener caracteres NUL intercalados. Esto es así porque los arrays de Ada siempre llevan asociados los límites (atributos 'First y 'Last).

Los literales de tipo String se encierran entre comillas.

Al ser String un array no restringido no podemos definir variables de ese tipo sin definir explícitamente o implícitamente los límites del array.

Nombre : String (1 .. 8); -- *Explícitamente* Nombre : String := “Fulanito"; -- *Implícitamente*

El operador & está definido para concatenar strings entre sí y caracteres con strings.

Nombre_Completo : **constant** String := Nombre & ' ' & Apellidos;

Puesto que hay dos tipos de caracteres, hay también dos tipos de strings: String y Wide_String que es un array de Wide_Character. En Ada 2005 aparece también el tipo Wide_Wide_String cuyos elementos son Wide_Wide_Character.

Para facilitar el uso de los strings hay varios paquetes predefinidos para su manejo:

- Ada.Strings.Fixed: para strings de tamaño fijo.
- Ada.Strings.Bounded: para strings con longitud variable y un limite superior definido.
- Ada.Strings.Unbounded: para strings con longitud variable y sin límites de tamaño.

Para manejo de Wide_Strings existen otros tres paquetes predefinidos que se nombran anteponiendo Wide_ a cada uno.

14.1 Manual de refencia de Ada

- 2.6 String Literals
- 3.6.3 String Types
- A.4.3 Fixed-Length String Handling
- A.4.4 Bounded-Length String Handling
- A.4.5 Unbounded-Length String Handling

Capítulo 15

Programación en Ada/Tipos/Registros

Un **registro** es un tipo que almacena varios campos de tipos distintos y que se identifican por su nombre. En C/C++ se llaman struct y en Pascal y Ada record.

type T_Sensor **is record** Magnitud : Natural; Alarma : Boolean; **end record**;

15.1 Acceso a los campos

Para acceder a los campos de un registro se usa la clásica notación registro.componente:

procedure Medir_temperatura **is** Sensor_temperatura : T_Sensor; **begin** Sensor_temperatura.Magnitud := 23; Sensor_temperatura.Alarma := False; -- ... **if** Sensor_temperatura.Alarma **then** Put_Line ("Alarma de temperatura"); **end if**; **end**;

Para asignar valores a todos los componentes se puede utilizar un agregado, lo cual es útil para asegurarnos de que no dejamos ninguno sin darle un valor:

procedure Asignar_temperatura **is** Sensor_temperatura : T_Sensor; **begin** Sensor_temperatura := (Magnitud => 23, Alarma => False); **end**;

También es posible dar un valor inicial a algunos o todos los componentes de un registro de modo que todos los objetos de ese tipo se inicialicen automáticamente con esos valores:

type T_Sensor **is record** Magnitud : Natural := 0; Alarma : Boolean := False; **end record**;

15.2 Registro nulo

El registro nulo se puede usar cuando se necesita un tipo que no almacene ningún dato (aunque parezca raro hay casos en los que es útil, como instanciar un genérico que pide un tipo, para el cual no tenemos ningún dato que pasarle). Hay dos maneras de declarar un registro nulo:

type Registro_Nulo **is record null**; **end record**; **type** Registro_Nulo **is null record**;

Ambas definiciones son semánticamente idénticas, la segunda es simplemente una sintaxis abreviada.

La declaración de una variable o constante de este tipo sería:

Nulo : **constant** Registro_Nulo := (**null record**);

15.3 Tipos especiales de registros

- Registros con discriminante
- Registros con parte variante
- Registros etiquetados, dan soporte de la programación orientada a objetos en Ada 95.

15.4 Manual de referencia de Ada

- 3.8 Record Types

Capítulo 16

Programación en Ada/Tipos/Registros discriminados

En un tipo **registro discriminado**, a algunos de los componentes se los conoce como discriminantes y el resto pueden depender de ellos. Los discriminantes tienen que ser de tipo discreto o puntero. Por ejemplo:

type T_Matriz **is array** (Integer **range** <>, Integer **range** <>) **of** Float; **type** T_Matriz_Cuadrada (Orden: Positive) **is record** Matriz: T_Matriz(1..Orden, 1..Orden); **end record**;

De esta forma, se asegura que la matriz utilizada sea cuadrada. El primer componente (Orden) es el discriminante del subtipo discreto Positive, mientras que el segundo componente (Matriz) es una matriz siempre cuadrada y cuyos límites dependen del valor de Orden. Ahora se utilizaría de la siguiente manera:

M: T_Matriz_Cuadrada(3); -- *También se puede emplear M: T_Matriz_Cuadrada(Orden => 3);* M := (3, (1..3 => (1..3 => 0.0))); M := (M.Orden, (M.Matriz'Range(1) => (M.Matriz'Range(2) => 5.0))); -- *En la sentencia anterior M.orden ya está definido e igual a 3.*

Una vez que se declara la variable ya no se puede cambiar su restricción. Sin embargo, si se declara lo siguiente:

Max: **constant** := 100; **subtype** T_Índice **is** Integer **range** 0..Max; **type** T_Vector_Enteros **is array** (Integer **range** <>) **of** Integer; **type** T_Polinomio (N: TÍndice := 0) **is** -- *Discriminante con valor inicial.* **record** Pol: T_Vector_Enteros(0..N); **end record**;

Ahora, se pueden declarar variables que no tengan restricciones:

P, Q: T_Polinomio;

El valor inicial de sus discriminantes sería cero (el valor por defecto de N). Así, se podría cambiar el discriminante posteriormente de esta manera:

P: T_Polinomio := (3, (5, 0, 4, 2)); R: T_Polinomio(5); -- *Aquí sólo se podrían usar polinomios de grado 5.*

16.1 Manual de referencia Ada

- 3.7 Discriminants

Capítulo 17

Programación en Ada/Tipos/Registros variantes

Puede interesar que un registro contenga partes de su estructura que puedan variar en función de otras. Para ello se definen los registros variantes que son un tipo especial de registros discriminados en los que la existencia de algunos campos depende del valor del discriminante.

Por ejemplo:

declare **type** TTipoVuelo **is** (MILITAR, CIVIL, ENEMIGO, DESCONOCIDO); **type** TRegistroVuelo (Clasif: TTipoVuelo) **is** **record** Velocidad: Float; Detalles: String(1..100); **case** Clasif **is** **when** CIVIL => **null**; **when** MILITAR => Origen, Destino: String(1..10); **when** ENEMIGO | DESCONOCIDO => NivelAmenaza: Integer; **end** **case**; **end** **record**; Vuelo1: TRegistroVuelo (MILITAR); **begin** Vuelo1.Origen := “Francia "; **end**;

Ilegal sería Vuelo1.NivelAmenaza := 1; pues dicho campo sólo es válido para DESCONOCIDO o ENEMIGO.

Si el discriminante lleva un valor por defecto entonces el registro es variante durante su ciclo de vida (es mutable), si no, el valor del discriminante se fija en su declaración y no varía durante la vida del objeto.

Los registros variantes en cierto modo están sobrepasados por la extensión de los tipos etiquetados, sin embargo aún son útiles en casos simples como aquellos en los que se conocen todas las posibilidades y no se desea que alguien añada una derivación a posteriori.

17.1 Véase también

- Registros
- Registros discriminados
- Registros etiquetados

17.2 Manual de referencia de Ada

- 3.8.1 Variant Parts and Discrete Choices

Capítulo 18

Programación en Ada/Tipos/Punteros a objetos

Un nombre está ligado a un objeto desde su declaración hasta que el flujo del programa deja la unidad que contenía su declaración. Sin embargo, los punteros o apuntadores (access) proporcionan acceso a otros objetos, que se pueden crear y destruir dinámicamente.

El nombre de *access* en vez del habitual *pointer* se debe a que al diseñar Ada se quería huir de la mala fama que los punteros habían creado gracias a lenguajes como C, en los que se puede usar los punteros de manera muy insegura. Los tipos acceso de Ada son más seguros entre otras cosas porque no existe la aritmética de punteros, especialmente peligrosa. Además el uso de punteros en Ada es prescindible en muchas más situaciones que en C.

Las variables de tipo puntero en Ada se inicializan implícitamente a null.

18.1 Ejemplos

Por ejemplo, se puede definir un puntero a un tipo entero de esta manera:

type PEntero **is access** Integer;

En un ejemplo con registros:

declare type TBúfer **is record** Mensaje: String(1..4); Prioridad: Integer; **end record**; **type** PTBúfer **is access** TBúfer; Mensaje1, Mensaje2: PTBúfer; **begin** Mensaje1 := **new** TBúfer; -- *Se crea un objeto de tipo TBúfer.* Mensaje2 := **new** TBúfer'(Prioridad => 2, Mensaje => “Hola”); -- *Con all se puede desreferenciar el puntero. -- Mensaje1 es un puntero y Mensaje1.all es el registro.* Mensaje1.**all**.Prioridad := 3; -- *Sin embargo, al acceder a campos del registro la desreferenciación -- puede hacerse implícita y .all es opcional en esos casos:* Mensaje1.Prioridad := 3; **end**;

Es útil para implementar listas, colas, árboles y grafos. Por ejemplo:

declare -- *TNodoÁrbolBinario se necesita para definir el puntero.* **type** TNodoÁrbolBinario; -- *Se declara después.* **type** PTNodoÁrbolBinario **is access** TNodoÁrbolBinario; **type** TNodoÁrbolBinario **is record** RamaIzda: PTNodoÁrbolBinario; Dato: Float; RamaDcha: PTNodoÁrbolBinario; **end record**; ÁrbolBinario: PTNodoÁrbolBinario; **begin** -- *Se crea la raíz del árbol binario.* ÁrbolBinario := **new** TNodoÁrbolBinario'(**null**, 1.0, **null**); **end**;

18.2 Liberación de memoria

Cuando se quiera liberar la memoria dinámicamente, hay que hacer uso del procedimiento genérico Ada.Unchecked_Deallocation , el cual se instancia con los tipos de objeto y de puntero, y se le llama pasándole punteros. Por ejemplo:

with Ada.Unchecked_Deallocation; **procedure** Ejemplo_Liberar_Memoria **is type** TVector **is array** (Integer **range** <>) **of** Float; **type** PVector **is access** TVector; PV: PVector; **procedure** Liberar_Vector **is new** Ada.Unchecked_Deallocation (TVector, PVector); **begin** PV := **new** TVector(1..10); PV.**all** := (others => 0.0); -- ... Liberar_Vector (PV); -- *La memoria es liberada y PV es ahora null* **end** Ejemplo_Liberar_Memoria;

El nombre de Unchecked_Deallocation viene del hecho de que no hay comprobación de que no queden punteros colgantes (*dangling pointers*), es decir que si se ha copiado el puntero en otra variable, después de llamar a Liberar_Vector el puntero copia está apuntando a una dirección de memoria no reservada y los efectos son imprevisibles, puesto que se puede haber reservado y se pude escribir o leer memoria que ya no pertenece a ese objeto.

Este sistema es similar al de C++ con new y delete. Un sistema de recolección de basura similar al de Java está previsto en el estándar, pero ningún compilador de Ada hasta el momento lo proporciona. Esto es debido a que aunque es un mecanismo más seguro, es menos eficiente y puede ser un problema para los sistemas de tiempo real por su impredictibilidad.

En Ada 95 existen métodos de gestión de memoria más seguros que el uso directo de Unchecked_Deallocation

basados en los tipos controlados, algo semejante a lo que se consigue en C++ con constructores y destructores que manejan memoria dinámica.

18.3 Manual de referencia de Ada

- 3.10 Access Types

Capítulo 19

Programación en Ada/Tipos/Punteros a subprogramas

Un puntero a subprograma nos permite llamar a un subprograma sin conocer su nombre ni dónde está declarado. Este tipo de punteros se suele utilizar en los conocidos *callbacks*.

type TPCallback **is access procedure** (Id : Integer; Mensaje : String); **type** TFCallback **is access function** (Mensaje : String) **return** Natural;

Para obtener el valor del puntero se usa el atributo 'Access aplicado a un subprograma con el prototipo adecuado, es decir, han de coincidir orden y tipo de los parámetros, y en el caso de las funciones, el tipo de retorno.

procedure ProcesarEvento (Id : Integer; Mensaje : String); MiCallback : TPCallback := ProcesarEvento'Access;

Los punteros a subprograma fueron una de las extensiones de Ada 95.

19.1 Manual de referencia de Ada

- 3.10 Access Types

Capítulo 20

Programación en Ada/Subtipos

Los subtipos se emplean para definir un subconjunto de un tipo determinado definido por una restricción.

Esta restricción puede ser un rango para un tipo escalar:

subtype TDíaDelMes **is** Integer **range** 1..31; **subtype** TDíaFebrero **is** TDíaDelMes **range** 1..29; **subtype** TLaborable **is** TDíaDeSemana **range** Lunes..Viernes;

O una restricción en un array irrestringido:

type TMatriz **is array** (Positive range <>, Positive **range** <>) **of** Integer; **subtype** TMatriz10x10 **is** TMatriz (1 .. 10, 1 .. 10);

O una restricción en el valor de un registro discriminado:

type TPersona (Sexo : TSexo) **is record** Nombre : TNombre; **case** Sexo **is when** Mujer => Embarazada : Boolean; **when** Hombre => **null**; **end case**; **end record**; **subtype** TMujer **is** TPersona (Sexo => Mujer);

Los subtipos de un mismo tipo base son totalmente compatibles entre sí, es decir no es necesaria una conversión de tipos para asignar objetos de subtipos distintos. Sin embargo, si en tiempo de ejecución se asigna un objeto a una variable y no se cumplen las restricciones del subtipo, se levantará la excepción Constraint_Error.

20.1 Manual de referencia de Ada

- 3.2 Types and Subtypes
- 3.2.2 Subtype Declarations

Capítulo 21

Programación en Ada/Tipos derivados

Hay ocasiones en las que es útil introducir un tipo nuevo que es similar en la mayoría de los aspectos a uno ya existente, pero que es un tipo distinto. Con la siguiente sentencia, se dice que S es un tipo derivado de T:

type S **is new** T;

21.1 Características

Los aspectos que definen a un tipo son su conjunto de valores y su conjunto de operaciones. El conjunto de valores de un tipo derivado es una copia del conjunto de valores del progenitor y, al ser copia, no pueden asignarse entre ambos. El conjunto de operaciones aplicables a un tipo derivado son:

- Los atributos son los mismos.
- A no ser que el tipo progenitor sea limitado, posee la asignación, igualdad y desigualdad predefinidas.
- El tipo derivado heredará los subprogramas primitivos del tipo progenitor, es decir, los subprogramas que tengan algún parámetro o un resultado de dicho tipo, y bien sean predefinidos o estén definidos en el mismo paquete que el tipo progenitor.

Estas operaciones heredadas, a efectos de uso, es como si estuvieran definidas en la misma región declarativa que el tipo derivado.

21.2 Ejemplo

Por ejemplo, si se tiene **type** TEnteroNuevo **is new** Integer, entonces el tipo derivado TEnteroNuevo heredará los subprogramas predefinidos como "+", "-", “abs”, etc. y los atributos como First o Last.

Así, por ejemplo, se puede escribir:

type TNumManzanas **is new** Integer; NumManzanas: TNumManzanas; -- ... NumManzanas := NumManzanas + 1;

Como puede observarse, se hace uso del operador "+" predefinido para el tipo Integer.

21.3 Tipos derivados frente a subtipos

¿Cuándo crear un tipo derivado en vez de un subtipo? Cuando queremos tener un nuevo tipo cuyos valores no deseamos que se confundan con los del tipo original pero queremos que posea las mismas operaciones y la misma representación que el tipo base. Por ejemplo:

-- Supongamos que tenemos cajas en las que caben 20 piezas de frutas **subtype** TNumFrutas **is** Natural **range** 1 .. 20; **type** TNumManzanas **is new** TNumFrutas; **type** TNumPeras **is new** TNumFrutas; NumManzanas: TNumManzanas; NumPeras: TNumPeras; NumFrutas: TNumFrutas; -- ... -- *Error de compilación, no podemos juntar peras con manzanas:* NumManzanas := NumManzanas + NumPeras; -- *Ilegal -- Pero siempre podemos contabilizarlos como frutas convirtiendo al tipo -- base:* NumFrutas := TNumFrutas (NumManzanas) + TNumFrutas (NumPeras);

Nota para programadores de C/C++: typedef realmente no define un tipo, sino que define un nuevo nombre para un mismo tipo, algo parecido al subtype, pero nada que ver con el **type** S **is new** T.

21.4 Manual de referencia de Ada

- 3.2.3 Classification of Operations
- 3.4 Derived Types and Classes

Capítulo 22

Programación en Ada/Tipos etiquetados

Ada 83 fue diseñado para dar soporte a desarrollos de sistemas empotrados, de tiempo real y misión crítica. Para este tipo de trabajos, las características de Ada, como el tipado fuerte y la división en paquetes, permitían el uso de una serie de metodologías de desarrollo, con un razonable grado de comodidad. Sin embargo, en los años 90, el advenimiento de la programación orientada a objectos, hizo necesario que Ada se extendiera para dar cabida a esta nueva metodología. Esto debido, a que la programación orientada a objeto necesitaba de un soporte en el manejo de tipos y en la visibilidad de los métodos, que Ada 83 hubiera considerado como no permitido.

Los cambios necesarios, fueron incorporados al nuevo estándar: Ada95.

Básicamente, los cambios se centraron alrededor del uso de los **tipos etiquetados**, una construcción software que añade flexibilidad al tipado fuerte, justo en grado necesario para permitir representar en Ada un **diagrama de objectos**, por ejemplo, en UML.

Los **tipos etiquetados** de Ada 95 son los que permiten realizar el polimorfismo. A menudo se asimila la combinación de un tipo etiquetado y el paquete que lo contiene a una clase en otros lenguajes de programación también orientados a objetos. Sin embargo hay algunas diferencias de sintaxis, que no de filosofía general, que veremos en este apartado.

Un tipo etiquetado puede ser un registro de modo que su estructura es pública, o puede ser un tipo privado, aunque en la parte privada siempre se termina definiendo como un registro.

type T_Base **is tagged record** ... **end record**; **type** T_Base **is tagged private**;

Un tipo etiquetado se puede extender.

type T_Derivado **is new** T_Base **with record** ... **end record**; **type** T_Derivado **is new** T_Base **with private**;

El nuevo tipo hereda todos las operaciones primitivas del tipo base y las puede redefinir.

Todo objeto de un tipo etiquetado contiene una etiqueta (*tag*) que permite reconocer su tipo en tiempo de ejecución.

22.1 Tipos polimórficos (class-wide type)

En Ada el polimorfismo se consigue con un tipo especial que puede contener objetos de cualquier tipo derivado de uno dado. Estos tipos especiales se indican con el atributo Class, es decir, T_Base'Class puede almacenar cualquier objeto derivado del tipo T_Base.

Este tipo de la clase es un tipo irrestringido, es decir, para declarar un objeto de este tipo tenemos que inicializarlo llamando a una función, asignándole otro objeto o con un *new* (si es un puntero a tipo polimórfico, es decir, access all T_Base'Class).

22.2 Llamadas que despachan

La ventaja del polimorfismo es poder escribir algoritmos que no dependen del tipo concreto de los objetos que se manejan.

El polimorfismo dinámico que proporciona la programación orientada a objetos permite que una operación sobre un objeto polimórfico se realice de distinto modo dependiendo del tipo concreto del objeto de que se trate en la ejecución. Otras formas de polimorfismo son el polimorfismo paramétrico que se implementa en Ada con las unidades genéricas y el polimorfismo por sobrecarga.

En Ada este polimorfismo se consigue llamando a una operación primitiva de un objeto polimórfico. Cuando el objeto sobre el que se realiza la operación es polimórfico y el parámetro formal del subprograma llamado es concreto, el subprograma realmente llamado puede ser uno redefinido para el tipo concreto si se ha redefinido para él una implementación distinta de la del tipo base.

Nota para programadores de C++: esto quiere decir que en Ada todos los "métodos" son virtuales, pero no todas las llamadas son virtuales, sólo las que llaman a un método de un objeto polimórfico (semejante a un puntero a clase base).

22.3 Manual de referencia de Ada

- 3.9 Tagged Types and Type Extensions
- 3.9.1 Type Extensions
- 3.9.2 Dispatching Operations of Tagged Types

Capítulo 23

Programación en Ada/Diseño y programación de sistemas grandes

Los sistemas empotrados suelen ser grandes y complejos, formados por subsistemas relacionados, pero relativamente independientes. Algunos lenguajes ignoran el hecho de que los programas se construyen por partes, cada una de ellas compilada por separado y todas ellas enlazadas en una aplicación final. El resultado se convierte en aplicaciones monolíticas difíciles de mantener. Otros lenguajes, en contraste, parten del concepto de módulo y proporcionan mecanismos de encapsulamiento y abstracción que ayudan a programar sistemas grandes, ya que el trabajo del equipo de programación y posterior mantenimiento del sistema se ve facilitado. Uno de estos lenguajes es *Ada*, que está fuertemente fundamentado en la disciplina de la ingeniería del *software* por lo que es el lenguaje más apropiado en la programación de sistemas empotrados industriales grandes.

Ada asume la necesidad de la compilación separada y proporciona dos mecanismos para realizarla, uno ascendente y otro descendente:

- **El mecanismo descendente (descomposición)**: consiste en dividir un sistema complejo en componentes más sencillos. Es apropiado para el desarrollo de grandes programas coherentes que, son divididos en varias subunidades que pueden compilarse por separado. Las subunidades se compilan después que la unidad de la que forman parte.

- **El mecanismo ascendente (abstracción)**: consiste en la especificación de los aspectos esenciales de un componente, posponiendo su diseño detallado. Es apropiado para la creación de bibliotecas de programa en las que las unidades se escriben para uso general y, consecuentemente, se escriben antes que los programas que las vayan a utilizar.

El diseño de sistemas mediante módulos permite encapsular partes del sistema mediante interfaces bien definidas y permiten utilizar técnicas que facilitan el desarrollo de sistemas grandes como:

- Ocultación de información.
- Tipos abstractos de datos.
- Compilación separada.

Las unidades de programa en *Ada* son las siguientes:

- **Subprograma**: que define los algoritmos ejecutables. Los procedimientos y las funciones son subprogramas.

- **Paquete**: define una colección de entidades. Los paquetes son el principal mecanismo de agrupación de *Ada*.

- **Tarea**: define una computación que puede llevarse a cabo en paralelo con otras computaciones.

- **Unidades genéricas**: ayudan a realizar código reutilizable. Pueden ser subprogramas o paquetes.

- **Unidad protegida**: puede coordinar el acceso a datos compartidos en el procesamiento paralelo. Aparece en el estándar *Ada 95*.

En *Ada*, las unidades de compilación pueden ser:

- Especificaciones de subprogramas
- Especificaciones de paquetes
- Cuerpos de subprogramas o paquetes

Algunos compiladores pueden establecer ciertos requisitos para las unidades de compilación. Por ejemplo, GNAT en su configuración predefinida exige que cada unidad esté definida en un fichero, con el nombre de la unidad y la extensión .ads para especificaciones y .adb para cuerpos. El guión "-" se ha de utilizar en sustitución del punto "." para unidades hijas y subunidades.

Capítulo 24

Programación en Ada/Paquetes

Los **paquetes** exportan mediante una interfaz bien definida tipos, objetos y operaciones y permiten ocultar su implementación, lo que proporciona al programador tipos abstractos de datos y subprogramas de manera transparente.

Los paquetes de Ada proporcionan:

- Abstracción de datos.
- Encapsulamiento.
- Compilación separada y dependiente.

24.1 Especificación y cuerpo

El paquete consta de especificación (parte visible) y cuerpo (implementación que se oculta) y pueden compilarse por separado.

La sintaxis de su especificación es la siguiente:

especificación_paquete ::= **package** [identificador_unidad_padre .] identificador **is** { declaración_básica } [**private** { declaración_básica }] **end** [[identificador_unidad_padre .] identificador] ;

La sintaxis del cuerpo de un paquete es la siguiente:

cuerpo_paquete ::= **package body** [identificador_unidad_padre .] identificador **is** [parte_declarativa] [**begin** secuencia_de_sentencias [**exception** manejador_de_excepción { manejador_de_excepción }]] **end** [[identificador_unidad_padre .] identificador] ;

Un paquete permite agrupar declaraciones y subprogramas relacionados.

24.2 Ejemplos

Por ejemplo, para implementar una pila de enteros:

package Pila_Enteros **is** -- Especificación. **procedure** Poner (Elem: Integer); -- Interfaz. **function** Quitar **return** Integer; -- Interfaz. **end** Pila_Enteros; **package body** Pila_Enteros **is** -- Cuerpo. Max : **constant** := 100; -- Se ocultan las variables locales. Pila: **array** (1..Max) **of** Integer; -- Se ocultan las variables locales. Cima: Integer **range** 0..Max; -- Se ocultan las variables locales. **procedure** Poner (Elem: Integer) **is** -- Implementación. **begin** Cima := Cima + 1; Pila (Cima) := Elem; **end** Poner; **function** Quitar **return** Integer **is** -- Implementación. **begin** Cima := Cima - 1; **return** Pila(Cima + 1); **end** Quitar; **begin** Cima := 0; -- Inicialización. **end** Pila_Enteros;

En este caso, se tiene una interfaz que proporciona acceso a dos subprogramas para manejar la pila, aunque también se podrían haber exportado tanto tipos como objetos, constantes, tareas e incluso otros paquetes. Por ejemplo:

package Sobre_Dias **is type** TDia **is** (LUN, MAR, MIE, JUE, VIE, SAB, DOM); **subtype** TDiaLaborable **is** TDia **range** LUN..VIE; SiguenteDia: **constant array** (TDia) **of** TDia := (MAR, MIE, JUE, VIE, SAB, DOM, LUN); **end** Sobre_Días;

En este caso, el paquete no necesitaría cuerpo. Todos los elementos definidos en el paquete son accesibles, por lo que podríamos utilizar Sobre_Dias.TDia, Sobre_Dias.SAB, Sobre_Dias.JUE o Sobre_Dias.TDiaLaborable.

24.3 Dependencia entre especificación y cuerpo

La especificación del paquete y el cuerpo pueden compilarse por separado. Mediante este encapsulamiento, ahora no es posible desde fuera modificar, por ejemplo, el valor de la cima de la pila, pues este objeto no es visible. Así se evita un mal empleo de la pila por alguien que pueda no conocer su implementación.

Si la especificación de un paquete contiene la especificación de un subprograma, entonces, el cuerpo del paquete debe contener el correspondiente cuerpo del subprograma. Sin embargo, pueden existir subprogramas dentro del cuerpo del paquete que no se declaren en su especificación, serían, por tanto, internos.

Destacar que dentro del cuerpo del paquete se inicializa el valor de la cima de la pila (después de begin). Esto su-

cede cuando se elabora el paquete. Si no necesita ninguna sentencia, se puede omitir el begin.

24.4 Declaración y visibilidad

Los paquetes se pueden declarar en cualquier parte declarativa, es decir, en un bloque, subprograma o dentro de otro paquete. En el caso del ejemplo, para utilizar la pila de números enteros, se podría hacer así:

declare N: Integer; **package** Pila_Enteros **is** -- ... **end** Pila_Enteros; **begin** Pila_Enteros.Poner (15); N := Pila_Enteros.Quitar; **end**;

Dentro del paquete se puede llamar a Poner o a Pila_Enteros.Poner, pero fuera del paquete únicamente se puede llamar a dicho procedimiento de la forma Pila_Enteros.Poner. Además, las variables Max, Pila y Cima no son visibles.

24.5 Importación de paquetes

Para la utilización de los paquetes desde otras unidades de compilación, se definen estas dos cláusulas:

- Cláusula use
- Cláusula with

24.6 Manual de referencia de Ada

- Section 7: Packages

Capítulo 25

Programación en Ada/Cláusula use

25.1 Definición

Si no se desea tener que escribir siempre Pila_Enteros.Poner para llamar a dicho procedimiento desde fuera del paquete, se puede utilizar la cláusula use, cuya sintaxis es la siguiente:

cláusula_use_paquete ::= **use** identificador { , identificador } ;

Así pues, siguiendo con el ejemplo de la sección anterior, se podría escribir:

-- ... **declare use** Pila_Enteros; N: Integer; **begin** Poner (15); N := Quitar; **end**;

Dicha cláusula use es semejante a una declaración y su ámbito llega al final del bloque. Incluso podría existir otra cláusula use más interna que se refiera al mismo paquete.

En el caso de existir varios paquete anidados, se puede utilizar la notación punto para distinguirlos, por ejemplo:

package P1 **is package** P2 **is** -- ... **end** P2; -- ... **end** P2; **use** P1, P1.P2; -- *Ilegal sería "use P2;"*

Para utilizar el paquete Standard, que contiene todas las entidades predefinidas, no se necesita cláusula use.

25.2 Desventajas

Muchos proyectos prohíben el uso de esta cláusula porque dificulta la comprensión del código y la localización de los tipos o subprogramas importados. Si deseas usarla lo más recomendable es usar un único use por cada unidad y haciéndolo sólo de los paquetes más conocidos o de los predefinidos.

25.3 Véase también

- Cláusula with

Capítulo 26

Programación en Ada/Cláusula with

Si una unidad se compila aparte (método usual si se va a utilizar en más programas), se puede hacer referencia a dicha unidad (cuya especificación y cuerpo pueden estar en ficheros distintos) mediante la **cláusula with**, cuya sintaxis es la siguiente:

cláusula_with ::= **with** identificador { , identificador } ;

Así, se podría utilizar el paquete Pila_Enteros del ejemplo anterior de esta manera:

with Pila_Enteros; -- *Se hace visible el paquete para todo el programa.* **procedure** Prueba_Pila_Enteros **is use** Pila_Enteros; -- *Para no usar nombre del paquete en las llamadas.* N: Integer; **begin** Poner (15); N := Quitar; **end** Prueba_Pila_Enteros;

La cláusula with tiene que ir antes de la unidad (no puede ir dentro de un ámbito más interno), de este modo, se ve claramente la dependencia entre las unidades.

Si la cláusula use se coloca inmediatamente después de la cláusula with correspondiente, se podrá utilizar los nombres de las entidades de los paquetes sin hacer referencia cada vez al nombre del paquete, en toda la unidad.

Si el paquete P usa los servicios del paquete Q y éste a su vez utiliza los servicios de R, a no ser que P utilice directamente los servicios de R, se debería utilizar únicamente la cláusula with Q; dentro de P. El programador de Q no tiene porqué conocer el paquete R.

Destacar que las especificaciones de unidades utilizadas con with deben compilarse antes que la unidad que contiene dichas cláusulas with, aunque de esto se suelen encargar automáticamente los compiladores.

También hay que tener en cuenta, que las dependencias que sean únicamente del cuerpo no deben darse en la especificación a no ser que se exporte algo que necesite de dicho paquete.

Hay un paquete que no es necesario que se mencione en la cláusula with, el paquete Standard.

26.1 Manual de referencia de Ada

- 10.1.2 Context Clauses - With Clauses

Capítulo 27

Programación en Ada/Declaraciones

Una **declaración** es una construcción del lenguaje que asocia un nombre con una entidad. *Ada* distingue cuidadosamente entre declaraciones (que introducen nuevos identificadores) y sentencias (que utilizan dichos identificadores). Hay dos clases de declaraciones:

- **Implícitas**: que ocurren como consecuencia de la semántica de otra construcción.
- **Explícitas**: aparecen en el texto del programa y pueden ser:
  - Declaraciones de tipo, subtipos, variables y constantes (objetos), excepciones, especificaciones de subprogramas o paquetes, cláusulas de representación o cláusulas use.
  - Los cuerpos de los subprogramas, paquetes y tareas.

27.1 Declaraciones de subprogramas

En ocasiones es necesario utilizar declaraciones de subprogramas, por ejemplo, si se va a utilizar la recursividad entre dos procedimientos:

procedure P; -- *Declaración de P, necesaria para utilizara en Q.* **procedure** Q **is** -- *Cuerpo de Q.* **begin** P; -- ... **end** Q; **procedure** P **is** -- *Repite especificación para declarar el cuerpo.* **begin** Q; -- ... **end** P;

También puede resultar útil declarar las especificaciones de los subprogramas en el comienzo del programa a modo de índice, sobre todo si hay muchos cuerpos.

27.2 Vista de una entidad

Todas las declaraciones contienen una definición de vista de una entidad. Una vista consiste en:

- Un identificador de la identidad.
- Características específicas de la vista que afectan al uso de la entidad a través de dicha vista.

En la mayoría de los casos, una declaración contiene la definición de la vista y de la entidad misma, pero una declaración de renombrado, sin embargo, no define una entidad nueva, sino que define una nueva vista de una entidad.

27.3 Parte declarativa

Una secuencia de declaraciones constituye una parte declarativa. Las siguientes construcciones de *Ada* tienen asociada una parte declarativa:

- Bloque.
- Cuerpo de un subprograma.
- Cuerpo de un paquete.
- Cuerpo de una tarea.
- Cuerpo de una entrada a un objeto protegido.

Por ejemplo, en un procedimiento, la parte declarativa comprendería las sentencias entre procedure y begin. El cuerpo de un procedimiento puede contener en su parte declarativa el cuerpo de otro procedimiento. Por tanto, los procedimientos pueden ser declarados sin límite de niveles de anidamiento sucesivos.

En ocasiones, las declaraciones requieren una segunda parte (se dice que requieren una terminación). La declaración y su terminación deben tener el mismo nombre y deben ocurrir en la misma región declarativa. Por ejemplo, un paquete necesita un cuerpo (que sería la terminación) si en su parte declarativa contiene alguna declaración que requiere una terminación que no se encuentre en dicha declaración.

27.4 Región declarativa de una declaración

Hay que distinguir entre región declarativa de una declaración y parte declarativa. En el supuesto de una declaración de una variable (por ejemplo I: Integer := 0;) se extiende por una región de texto que abarca sólo una línea, sin embargo, otras declaraciones más complejas como procedimientos y paquetes constituyen muchas más líneas de texto. En *Ada*, existen las siguientes construcciones que tienen asociada una región declarativa:

- Cualquier declaración que no sea terminación de otra.
- Bloque
- Bucle
- Construcción accept.
- Manejador de excepción.

La región declarativa de cada una de estas construcciones comprende:

- El texto de la construcción misma.
- Texto adicional, determinado de esta manera:
  - Si una declaración está incluida en la región, también lo está su terminación.
  - Si una unidad de biblioteca está incluida en la región, también lo están sus unidades hijas.
  - Si está incluido el requerimiento de compilación separada, también lo está su subunidad correspondiente.

Así, se pueden extraer las siguientes conclusiones:

- La especificación de un paquete y su cuerpo forman parte de la misma región declarativa porque el cuerpo es la terminación de la especificación, por lo que no se puede declarar la misma variable en la especificación y en el cuerpo.

- Ya que la declaración y el cuerpo de un paquete pueden estar en compilaciones distintas, la región declarativa de una declaración de paquete puede abarcar porciones de texto en varias unidades de compilación distintas.

- Todas las unidades de biblioteca son hijas de Standard, luego toda unidad de biblioteca pertenece a su región declarativa.

27.5 Manual de referencia de Ada

- 3.1 Declarations

Capítulo 28

Programación en Ada/Ámbito

Cada declaración tiene asociada una porción del texto del programa que se denomina **ámbito** de la declaración. En dicha porción es el único lugar donde se puede hacer referencia a dicha declaración. El ámbito de una declaración consiste en:

- Si la declaración es una unidad de biblioteca, todos sus dependientes semánticos.
- Si la declaración no es una unidad de biblioteca, una porción de la región declarativa que la encierra inmediatamente. Dicha porción se extiende desde el comienzo de la declaración hasta el final de su región declarativa.

Resumiendo, como ejemplo:

procedure P **is** -- *////////////////////////////////// Parte declarativa de P.* -- ... -- *----------* *Ámbito de A.* A: Float; -- *----------* -- *Ámbito de Q.* -- *////////////// Región declarativa de I y R.* **procedure** Q **is** -- *//////////////////////////// Parte declarativa de Q.* -- *---------- Ámbito de I.* I: Integer := 0; -- ... -- *//////// Región declarativa de J.* **package** R **is** -- *//////// Parte declarativa pública de R.* -- *----------* *Ámbito de J.* J: Integer := I; -- ... -- *///// Fin parte declarativa pública de R.* **end** R; **package body** R **is** -- *//////// Parte declarativa privada de R.* -- *----------* *Ámbito de K* K: Integer := I + J; -- ... -- *///// Fin parte declarativa privada de R.* **begin** -- ... **end** R; -- *----------* *Fin ámbito de K* -- *---------- Fin ámbito de J.* -- *//// Fin región declarativa de J.* -- *////////////////////////* *Fin parte declarativa de Q.* **begin** -- ... **end** Q; -- ... -- *----------* *Fin ámbito de I.* -- *////////// Fin región declarativa de I y R.* -- *////////////////////////////// Fin parte declarativa de P.* **begin** -- ... **end** P; -- *----------* *Fin ámbito de A.* -- *----------* *Fin ámbito de Q.*

Para la cláusula with también se define un ámbito, que consiste en la región declarativa de su declaración si aparece delante de la declaración de una unidad de biblioteca y en el cuerpo si aparece delante de un cuerpo.

Para la cláusula use: si actúa como una declaración, su ámbito es la porción de la región declarativa que empieza justo después de la cláusula y finaliza junto con dicha región declarativa; si actúa como cláusula de contexto, su ámbito es el mismo que el de la cláusula with.

28.1 Manual de referencia de Ada

- 8.2 Scope of Declarations

Capítulo 29

Programación en Ada/Visibilidad

Una entidad es **visible** en un punto dado si se puede utilizar su identificador para referirse a ella en dicho punto. La diferencia con el ámbito es que éste es la región de texto donde una determinada entidad es visible.

En el caso de una declaración de una variable, no se puede utilizar el identificador hasta que no se haya terminado su declaración (por ejemplo, sería ilegal la declaración I: Integer := I +1;) ya que no es visible hasta que no se haya terminado de declarar.

Como ejemplo, en el caso de un bloque:

declare -- *Ámbito de I externa.* I, J: Integer; -- *Visibilidad de I externa.* **begin** -- ... **declare** -- *Ámbito de I interna.* I: Integer := 0; -- *Visibilidad de I interna, oculta la visibilidad de I externa.* **begin** -- ... **end**; -- *Fin visibilidad de I interna, oculta la visibilidad de I externa. -- Fin ámbito de I interna.* **end**; -- *Fin visibilidad de I externa. -- Fin ámbito de I externa.*

En este caso, la visibilidad de la I externa se ve ocultada cuando se hace visible a la I interna. Sin embargo, si se dota de nombre al bloque, se puede hacer referencia a una variable *supuestamente ocultada* con la notación punto:

Externo: **declare** I, J: Integer; -- *I externa.* **begin** -- ... **declare** I, K: Integer; -- *I interna.* **begin** K := J + Externo.I; -- *Se hace referencia a la I externa.* **end**; **end** Externo;

Igualmente se puede hacer con bucles y, por supuesto, con subprogramas y paquetes, pues deben poseer un identificador.

En el caso de una entidad declarada en la parte no privada de la especificación de un paquete, se aplican las mismas reglas dentro del paquete pero, fuera del mismo, la entidad no es visible a menos que se escriba el nombre del paquete mediante la notación punto o, alternativamente, se escriba una cláusula use.

29.1 Reglas de visibilidad

Una declaración es directamente visible en un lugar determinado cuando su nombre, si notación punto, es suficiente para referenciarla. La visibilidad puede ser inmediata o mediante una cláusula use.

Los identificadores visibles en un punto son aquellos visibles antes de considerar ninguna cláusula use más aquellos que se hacen visibles debido a las cláusulas use.

La regla básica es que un identificador de un paquete se hace visible mediante una cláusula use si el mismo identificador no está también en otro paquete con otra cláusula use, por supuesto, siempre que el identificador no sea ya visible. Si no se cumple, hay que recurrir a la notación punto.

Se aplica una regla ligeramente diferente cuando todos los identificadores son subprogramas o literales enumeración. En este caso, un identificador que se hace visible mediante una cláusula use no puede ocultar nunca otro identificador, aunque sí puede sobrecargarlo.

29.2 Manual de referencia de Ada

- 8.3 Visibility

Capítulo 30

Programación en Ada/Renombrado

El **renombrado** o redenominación se utiliza para dar a una entidad un identificador más conveniente en una determinada porción del programa. Se suele emplear para resolver ambigüedades y para evitar el uso de la notación punto. Para ello se emplea la palabra reservada **renames**. Por ejemplo:

function "*" (X, Y: TVector) **return** Float **renames** ProductoEscalar;

Con ello se consigue utilizar indistintamente tanto "*" como ProductoEscalar (definido con anterioridad) para referirse a la misma función.

También se puede evitar la notación punto sin tener que importar todos los identificadores con la cláusula use:

procedure Poner (Elem: Integer) **renames** PilaEnteros.Poner;

El renombrado se puede utilizar con objetos (variables y constantes), excepciones, subprogramas, y paquetes. No se aplica a tipos, aunque un subtipo que no añade restricciones es equivalente a un renombrado.

F: TFecha **renames** Agenda(I).FechaNacimiento; ------ **package** P **renames** Plantilla_Pila;

Reseñar que el renombrado no corresponde a una sustitución de texto. La identidad del objeto se determina cuando se realiza el renombrado.

30.1 Manual de referencia de Ada

- 8.5 Renaming Declarations

Capítulo 31

Programación en Ada/La biblioteca

31.1 La biblioteca *Ada* (unidades y subunidades)

La biblioteca *Ada* es la piedra angular de este lenguaje en la construcción de sistemas grandes, pero fiables.

Los programas grandes deben ser descompuestos en subsistemas, cada uno de ellos con su propia estructura interna. La respuesta a este requerimiento en *Ada* es la biblioteca *Ada* y tiene las siguientes características:

- Integrado en el lenguaje.
- Facilita la creación y mantenimiento de un subsistema, actuando como repositorio estructurado de todos sus componentes.
- Ofrece a los programas que hacen uso de un subsistema un interfaz fácil de utilizar y que es selectivo a componentes internos.

Los compiladores de *Ada* toman el código fuente y la biblioteca referenciada y producen un código objeto y, además, una biblioteca actualizada con dicho código objeto. Es como si la biblioteca *Ada* "recordara" las compilaciones que se realizan en el sistema. A diferencia de los compiladores de otros lenguajes, que únicamente generan el código objeto sin incorporarlo a ninguna biblioteca.

El concepto de incorporación a la biblioteca no está definido por el lenguaje *Ada*, sino por el propio compilador. Por ejemplo, en la implementación de *Ada* de *GNU* denominada *GNAT*, la biblioteca se implementa sobre un sistema de ficheros. La compilación de un fichero que contiene, por ejemplo, un procedimiento, produce un fichero objeto y una colección de enlaces al resto de la biblioteca (fichero con la extensión .ali de *Ada Library Information*), dentro del mismo directorio. El compilador puede tener ahora dos "vistas" diferentes de la biblioteca *Ada*, una con el procedimiento incorporado y otra sin él.

La estructura formal de un programa Ada es la siguiente:

- Un **programa** es un conjunto de compilaciones. El concepto de compilación no está especificado por el lenguaje Ada, pero suele ser un fichero fuente.
- Una **compilación** es una secuencia de unidades de compilación. Por ejemplo, una compilación con seis unidades de compilación puede ser un fichero con cuatro procedimientos y dos paquetes. El número de unidades de compilación en una compilación puede estar limitado por la implementación. Por ejemplo, el compilador *GNAT* únicamente permite una unidad de compilación por cada compilación.
- Una **unidad de compilación** puede ser bien una unidad de biblioteca o bien una subunidad.
- Una **unidad de biblioteca** es la declaración o cuerpo de un procedimiento o de un paquete.
- Una **subunidad** es una parte de una unidad de biblioteca que se desea separar y compilar por separado.

La biblioteca se alimenta de los programas, que no son más que un conjunto de unidades de compilación que se suman a la biblioteca cuando se compila el programa. Cuando un programa está correctamente construido, se incorpora a la biblioteca. Los programas nuevos utilizan el material compilado ya disponible en la propia biblioteca.

Hay que tener presente que los programas se escriben por partes que son compiladas por separado y luego se enlazan para dar el resultado final. *Ada* proporciona dos mecanismos para ello:

- Unidades de biblioteca: mecanismo ascendente.
- Subunidades: mecanismo descendente.

31.1.1 Subsecciones

1. Unidades de biblioteca
2. Unidades hijas
3. Subunidades
4. Compilación separada y dependiente

Capítulo 32

Programación en Ada/Unidades de biblioteca

Una **unidad de biblioteca** puede ser una especificación de subprograma o una especificación de paquete; a los cuerpos correspondientes se les denomina unidades secundarias. Se puede compilar especificación y cuerpo juntos, pero es conveniente hacerlo separadamente con el fin de mejorar la accesibilidad y el mantenimiento de los programas.

Cuando se compila una unidad, ésta se almacena dentro de la biblioteca de programas. Una vez que se incluye en la biblioteca, una unidad puede ser usada por cualquier otra unidad que se compile a continuación, esta dependencia se indica con la cláusula with.

Si el cuerpo de un subprograma es por sí mismo suficiente para definir un subprograma completo. Es entonces cuando se le clasifica como una unidad de biblioteca, en vez de tratarlo como una subunidad.

Si la especificación y el cuerpo se compilan por separado, entonces, el cuerpo debe compilarse después de la especificación, es decir, el cuerpo es dependiente de la especificación. Sin embargo, toda unidad que utilice el paquete es dependiente únicamente de la especificación, aspecto destacable de Ada. Con ello, aunque cambie el cuerpo del paquete, si no se cambia la especificación (interfaz con el exterior), no es necesario volver a recompilar las unidades que estaban utilizando dicho paquete. Se puede apreciar que la compilación separada de especificación y cuerpo simplifica el mantenimiento de los programas.

Como es obvio, las unidades de biblioteca no pueden sobrecargarse ni pueden ser operadores.

32.1 Manual de referencia de Ada

- 10.1.1 Compilation Units - Library Units

Capítulo 33

Programación en Ada/Unidades hijas

El empleo de **unidades hijas** surge ante la necesidad de poder referenciar a un gran número de unidades de biblioteca con distintos nombres. Al igual que un sistema de archivos jerarquizado mediante directorios y subdirectorios, la biblioteca Ada contiene una jerarquía en su organización. Las unidades hijas son un paso más allá respecto a las subunidades, pues permiten extender la funcionalidad de un paquete sin modificar el paquete en cuestión.

El padre de todas las unidades de biblioteca es el paquete Standard. De este modo, las unidades de biblioteca creadas se agregan como hijas de Standard y se les denomina unidades de biblioteca raíz. Estas unidades serían hermanas de los paquetes predefinidos Standard.Ada, Standard.System y Standard.Interfaces. Y cada una de ellas puede a su vez contener unidades hijas.

33.1 Espacio de nombres

Las unidades hijas forman también un espacio de nombres, desde dentro de la jerarquía es posible referirse a las entidades definidas en el paquete padre como si estuviesen en la propia unidad. Igualmente al referirse a unidades hermanas.

Por ejemplo:

package Servidor **is type** Petición_T **is private**; -- ... **end** Servidor; **package** Servidor.Sesión **is type** Sesión_T **is record** Petición : Petición_T; -- *Equivalente a Servidor.Petición_T* -- ... **end record**; -- ... **end** Servidor.Sesión;

Los paquetes se pueden encontrar en cualquier punto de la jerarquía, y los subprogramas sólo en las hojas del árbol.

33.2 Visibilidad

La parte privada de un paquete hijo y su cuerpo pueden referenciar las entidades definidas en la parte privada del paquete padre.

33.3 Manual de referencia de Ada

- 10.1.1 Compilation Units - Library Units

Capítulo 34

Programación en Ada/Subunidades

El cuerpo de un paquete, subprograma o tarea puede ser "extraído" de la unidad o subunidad de biblioteca que lo engloba y compilarse por separado en lo que viene a denominarse subunidad. En la unidad que lo engloba, el cuerpo "extraído" se sustituye por un "resguardo" del cuerpo. Cualquier unidad de compilación puede tener subunidades.

En un ejemplo anterior, se construía un paquete de una pila de números enteros con dos procedimientos Poner y Quitar, que interesa compilar por separado, luego se escribiría:

package body Pila_Enteros **is** -- *Cuerpo.* Max : **constant** := 100; Pila: array(1..Max) **of** Integer; Cima: Integer **range** 0..Max; **procedure** Poner (Elem: Integer) **is separate**; -- *Se compila aparte.* **function** Quitar **return** Integer **is separate**; -- *Se compila aparte.* **begin** Cima := 0; -- *Inicialización.* **end** Pila_Enteros;

A los subprogramas que se van a compilar aparte (Poner y Quitar) se les denomina subunidades. Su cuerpo deberá implementarse en otro fichero de esta forma:

separate (Pila_enteros) -- *Indica la unidad de la que se extrajo.* **procedure** Poner (Elem: Integer) **is begin** Cima := Cima + 1; Pila (Cima) := Elem; **end** Poner;

Y de manera análoga se procedería con Quitar.

En el caso de que R sea subunidad de Q y ésta a su vez de P, que es una unidad de biblioteca, entonces la implementación de R debe comenzar con separate (P.Q).

Una subunidad depende de la unidad de la que fue separada y, por tanto debe compilarse después de ella.

La visibilidad dentro de la subunidad es exactamente igual que si no hubiera sido separada, es decir, por ejemplo, una cláusula with en la unidad principal se aplica a todas sus subunidades.

Si se necesita de una unidad únicamente dentro de una subunidad, a fin de no complicar las dependencias de compilación, se deberá incluir la cláusula with justo antes de la declaración subunidad, es decir, delante de separate (Pila_Enteros).

34.1 Manual de referencia de Ada

- 10.1.3 Subunits of Compilation Units

Capítulo 35

Programación en Ada/Compilación separada y dependiente

Ada realiza una **compilación separada y dependiente**.

Una compilación separada significa que el programa principal y un subprograma pueden escribirse por separado en ficheros distintos.

Una compilación dependiente significa que el compilador va a llevar a cabo la comprobación de que los tipos y el número de parámetros de la invocación en el subprograma invocante concuerdan con los tipos y el número de parámetros del subprograma invocado.

En otros lenguajes en los que se realiza una compilación independiente (por ejemplo el lenguaje *C*), no se advierte que los parámetros de llamada se corresponden y compila correctamente. Esta situación en un sistema de control es intolerable. El fallo no se detecta en la compilación y puede que tampoco en las pruebas.

En *Ada*, cuando desde una unidad de biblioteca se utiliza un tipo o un subprograma de otra unidad, se puede entender que depende semánticamente de ella.

Cuando una unidad ha sido compilada con éxito, se incorpora a la biblioteca del lenguaje. Así, cuando el compilador encuentra una llamada a un subprograma, contrasta el número y el tipo de los parámetros de la llamada contra la declaración del subprograma invocado, declaración que debe haber sido previamente compilada y que, en consecuencia, debe estar ya en la biblioteca. Por lo tanto, se puede decir que es la biblioteca *Ada* la que implementa la dependencia.

Ada permite incluso escribir y compilar la subrutina invocante antes que la subrutina invocada de forma consistente. Esto se consigue compilando únicamente la especificación, dejando la compilación del cuerpo para más tarde. En dicha especificación se deja detallado el nombre, el número y los tipos de los parámetros, además de indicar si son de entrada, salida o ambos. Esta es toda la información que necesita el compilador para compilar una llamada a un subprograma. Cuando, posteriormente, se compile el cuerpo del subprograma, se comprobará que es consistente con la especificación.

La forma de expresar que una unidad depende de otra se realiza mediante la cláusula with. Cuando el compilador encuentra dicha cláusula, extrae de la biblioteca el interfaz de la unidad que acompaña a with.

El orden de compilación es el siguiente: una unidad sólo se incorpora a la biblioteca después de que todas las unidades de las que depende se han incorporado también a la biblioteca. Ello implica que:

- Si especificación y cuerpo de una unidad se compilan por separado, es preciso compilar antes la especificación.
- Si la especificación de una unidad es cambiada y, por lo tanto, es recompilada de nuevo, todas las unidades que dependen de ella deben ser recompiladas.
- Si el cuerpo de una unidad se cambia de una forma consistente con su especificación, las unidades que dependen de esta unidad no necesitan ser recompiladas.

El lenguaje *Ada* viene con varios paquetes predefinidos como Text_IO. Estos paquetes ya han sido incorporados a la biblioteca del lenguaje. Hay, sin embargo, una excepción que es el paquete Standard, que no necesita la cláusula with. Finalmente, todas las unidades incorporadas a la biblioteca *Ada* deben tener nombres diferentes. En otro caso, se produce el reemplazamiento de la unidad residente por la nueva unidad con el mismo nombre.

Notese que cuando se dice que una unidad se ha de compilar antes que otra no quiere decir que el programador se tenga que preocupar de estos temas, pues los entornos de desarrollo de Ada vienen acompañados de herramientas de compilación que se encargan de recompilar todas las unidades necesarias y sólo las que han quedado obsoletas por un cambio en el código fuente. Por ejemplo, con el compilador GNAT, esta herramienta es gnatmake.

35.1 Manual de referencia de Ada

- Section 10: Program Structure and Compilation Issues

Capítulo 36

Programación en Ada/Tipos abstractos de datos

36.1 Tipos abstractos de datos (tipos privados)

Una de las principales contribuciones de los lenguajes de alto nivel es que el programador no tiene que preocuparse de cómo se representan físicamente los datos en el computador. De esta idea surge el concepto de tipo de datos. Una extensión del mismo es el tipo abstracto de datos. Su implementación es de nuevo desconocida para el programador, esta vez no porque desconozca la arquitectura del computador subyacente, sino porque es encapsulado en un módulo que no permite el acceso directo a los detalles de su implementación. En su lugar, se proporciona al programador operaciones sobre el tipo que son invocaciones a entradas del módulo que lo encapsula.

Por ejemplo, consideremos la utilización de un tipo abstracto de datos que represente a un número complejo:

package Números_complejos **is type** TComplejo **is record** Real, Imag: Float; **end record**; I: **constant** TComplejo := (0.0, 1.0); **function** "+" (X, Y: TComplejo) **return** TComplejo; **function** "-" (X, Y: TComplejo) **return** TComplejo; **function** "*" (X, Y: TComplejo) **return** TComplejo; **function** "/" (X, Y: TComplejo) **return** TComplejo; **end** Números_complejos;

De este modo, el usuario debe conocer los detalles de la implementación y sabe que se utiliza una representación cartesiana. Además, el usuario está obligado a hacer uso de la representación.

Para impedir el uso del conocimiento de la representación con vistas, por ejemplo, a poder cambiar ésta posteriormente, se puede hacer uso de los tipos privados definiéndolos mediante la palabra reservada **private**:

package Números_complejos **is** -- *Parte visible.* **type** TComplejo **is private**; -- *Tipo privado.* I: **constant** TComplejo; -- *No se puede asignar valor todavía.* **function** "+" (X, Y: TComplejo) **return** TComplejo; **function** "-" (X, Y: TComplejo) **return** TComplejo; **function** "*" (X, Y: TComplejo) **return** TComplejo; **function** "/" (X, Y: TComplejo) **return** TComplejo; **function** Construir_complejo (R, I: Float) **return** TComplejo; **function** Parte_imaginaria (X: TComplejo) **return** Float; **function** Parte_real (X: TComplejo) **return** Float; **private** -- *Parte oculta.* **type** TComplejo **is record** Real, Imag: Float; **end record**; I: **constant** TComplejo := (0.0, 1.0); **end** Números_complejos;

Ahora, se ha definido TComplejo como tipo privado y se resguardan los detalles de su implementación en la parte no visible del paquete después de la palabra reservada **private** y hasta el fin de la especificación del paquete. En la parte visible (desde el comienzo de la especificación hasta private), se da la información disponible fuera del paquete.

Las únicas operaciones disponibles son la asignación, la igualdad y la desigualdad, aparte de las añadidas en el paquete.

Nótese que el valor de I no se puede dar pues no se conocen todavía los detalles de la implementación, se declara como constante y se le asigna después un valor en la parte privada.

Las funciones Construir_complejo, Parte_imaginaria y Parte_real son ahora necesarias pues el usuario ya no conoce la estructura del tipo TComplejo y se necesita realizar dicha interfaz para poder manejar objetos del tipo privado.

El cuerpo se podría implementar de la siguiente manera:

package body Números_complejos **is function** "+" (X, Y: Tcomplejo) **return** TComplejo **is begin return** (X.Real + Y.Real, X.Imag + Y.Imag); **end** "+"; -- ... *"-", "* y "/" similarmente.* **function** Construir_complejo (R, I: Float) **return** TComplejo **is begin return** (R, I); **end** Construir_complejo; **function** Parte_real (X: TComplejo) **return** Float **is begin return** X.Real; **end** Parte_real; -- ... *Parte_imaginaria análogamente.* **end** Números_complejos;

Y podría ser utilizado transparentemente, por ejemplo, dentro de un bloque como:

declare use Números_complejos; C1, C2: TComplejo; R1, R2: Float; **begin** C1 := Construir_complejo (1.5, −6.0); C2 := C1 + I; R := Parte_real (C2) + 8.0; **end**;

Si ahora se quisiera cambiar la implementación del tipo TComplejo y representarlo en forma polar, no sería necesario cambiar la parte visible de la especificación, por lo que todas las unidades que utilicen dicho paquete no tienen la necesidad de actualizarse. La interfaz exportada no ha cambiado y, por tanto, los programas que la utilizarán pueden seguir haciéndolo. Por ejemplo, ahora se podría representar en la parte privada de la especificación del paquete como:

-- ... **private** Pi: **constant** := 3.1416; **type** TComplejo **is record** R: Float; Theta: Float **range** 0.0 .. 2*Pi; **end** recod; I: **constant** TComplejo := (1.0, 0.5*Pi); **end** Números_complejos;

Lo único que se necesitaría sería reescribir el cuerpo del paquete y recompilarlo.

36.2 Enlaces externos

- Ejemplos de TADs

36.2.1 Manual de referencia de Ada

- 7.3 Private Types and Private Extensions

Capítulo 37

Programación en Ada/Tipos limitados

37.1 Tipos privados limitados

Cuando se define un tipo privado, se predefinen inherentemente las operaciones de asignación, igualdad y desigualdad. Si no se quiere que exista ninguna operación, sino únicamente las definidas en el paquete, se debe emplear el tipo privado limitado.

Como consecuencia de no tener operador de asignación, la declaración de un objeto de dicho tipo no puede incluir un valor inicial. Esto también tiene la consecuencia de que no pueden existir constantes de un tipo privado limitado.

La ventaja es que el programador de la unidad que contenga un tipo privado limitado se asegura el control absoluto sobre los objetos de dicho tipo.

Para indicarlo, se define el tipo como **limited private**. Por ejemplo, implementado un tipo abstracto de datos pila:

package Pilas **is type** TPila **is limited private**; -- *Tipo privado limitado.* **procedure** Poner (P: **in out** TPila; X: **in** Integer); **procedure** Quitar (P: **in out** TPila; X: **out** Integer); **function** "=" (P1, P2: TPila) **return** Boolean; **private** Max: **constant** := 100; **type** TVectorEnteros **is array** (Integer **range** <>) **of** Integer; **type** TPila **is record** P: TVectorEnteros(1..Max); Cima; Integer **range** 0..Max := 0; **end record**; **end** Pilas;

La función "=" se implementa para comprobar que dos pilas tienen el mismo número de elementos y cada uno de ellos en el mismo orden deber ser iguales. Por eso, se ha optado por un tipo privado limitado.

37.2 Manual de referencia de Ada

- 7.5 Limited Types

Capítulo 38

Programación en Ada/Unidades genéricas

38.1 Polimorfismo paramétrico

La idea de reutilización de código surge ante la necesidad de construir programas en base a componentes bien establecidos que pueden ser combinados para formar un sistema más amplio y complejo. La reutilización de componentes mejora la productividad y la calidad del *software*. El lenguaje *Ada* soporta esta característica mediante las unidades genéricas.

Una unidad genérica es aquella en la que se manipulan tipos que posteriormente instanciará el usuario, es decir, se utiliza a modo de plantilla. Se pueden hacer unidades genéricas de subprogramas y paquetes. Sintácticamente se podría describir como:

unidad_genérica ::= **generic** { lista_parámetros_genéricos } (especificación_subprograma | especficicación_paquete) lista_parámetros_genéricos ::= identificador { , identificador } [**in** [**out**]] tipo [:= expresión] ; | **type** identificador **is** ((<>) | **range** <> | **digits** <> | **delta** <> | definición_vector | definición_puntero) | declaración_privada_de_tipo | declaración_formal_procedimiento | declaración_formal_paquete

Por ejemplo, para reutilizar un procedimiento de intercambio de variables:

generic type TElemento **is private**; -- Parámetro tipo formal genérico. **procedure** Intercambiar (X, Y: **in out** TElemento); **procedure** Intercambiar (X, Y: **in out** TElemento) **is** Temporal : TElemento; **begin** Temporal := X; X := Y; Y := Temporal; **end** Intercambiar;

La especificación del subprograma va precedida por la parte formal genérica, que consta de la palabra reservada generic seguida por una lista de parámetros formales genéricos que puede ser vacía.

El subprograma Intercambiar es genérico y se comporta como una plantilla. Hay que destacar que las entidades declaradas como genéricas no son locales, por ello es necesario instanciarlas. Por ello, para poder utilizar la unidad del ejemplo es necesario crear una instancia suya para el tipo que se quiera usar, su sintaxis sería:

instanciación_unidad_genérica ::= (**package** | **procedure** | **function**) identificador **is new** identificador [(parámetro_instanciado { , parámetro_instanciado })] ;

Por ejemplo:

procedure Intercambiar_enteros **is new** Intercambiar (Integer);

Con ello, se puede utilizar el procedimiento para tipos Integer, si se quiere utilizar para cualquier otro tipo basta con volver a instanciarlo para el nuevo tipo con otro nombre o, si se utiliza el mismo identificador en la instanciación, se sobrecarga el procedimiento y puede ser utilizado para distintos tipos:

procedure Inter **is new** Intercambiar (Float); **procedure** Inter **is new** Intercambiar (TDía); **procedure** Inter **is new** Intercambiar (TElemento => TPila);

De igual modo, se pueden emplear paquetes genéricos, por ejemplo, para implementar una plantilla del tipo abstracto de datos pila:

generic -- Especificación unidad genérica. Max: Positive; -- Parámetro bjeto formal genérico. **type** TElemento **is private**; -- Parámetro tipo formal genérico. **package** Plantilla_pila **is procedure** Poner (E: TElemento); **function** Quitar **return** TElemento; **end** Plantilla_pila; **package body** Plantilla_pila **is** -- Cuerpo unidad genérica. Pila: array(1..Max) **of** TElemento; Cima: Integer **range** 0..Max; -- ... **end** Plantilla_pila;

Ahora se podría utilizar una pila de un tamaño y tipo determinados, para ello, habría que crear un ejemplar, por ejemplo, de esta manera:

declare package Pila_reales_de_100 **is new** Plantilla_pila (100, Float); **use** Pila_reales_de_100; **begin** Poner (45.8); -- ... **end**;

En la segunda línea del código anterior se ha creado una instancia del paquete Plantilla_pila que se llama Pila_reales_de_100. A partir de ese momento se pueden acceder a los miembros del paquete, ya que la creación de la instancia implica su visibilidad (igual que si se ejecutara la sentencia **with** Pila_reales_de_100;).

38.2 Parámetros de unidades genéricas

Resaltar que los objetos declarados como parámetros formales son de modo **in** por defecto y pueden ser **in** o **in out**, pero nunca **out**. En el caso de que sea **in**, se comportará como una constante cuyo valor lo proporciona el parámetro real correspondiente. Como resulta obvio, un parámetro genérico **in** no puede ser de un tipo limitado, pues no se permite la asignación y el parámetro formal toma su valor mediante asignación. En el caso de que el parámetro genérico sea de modo **in out**, se comporta como una variable que renombra al parámetro real correspondiente; en este caso, el parámetro real debe ser el nombre de una variable y su determinación se realiza en el momento de la creación del ejemplar.

Además de parámetros genéricos de tipo y objetos, se pueden incluir parámetros formales de subprogramas o paquetes, por ejemplo:

generic type TElem **is private**; **with function** "*" (X, Y: TElem) **return** TElem; **function** cuadrado (X : TElem) **return** TElem; **function** cuadrado (X: TElem) **return** TElem **is begin return** X * X; -- El operador "*" formal. **end** cuadrado;

Se utilizaría, por ejemplo, con matrices (teniendo previamente definida la operación de multiplicación de matrices), de la siguiente manera:

with Cuadrado; **with** Matrices; **procedure** Prueba_operaciones **is function** Cuadrado_matriz **is new** Cuadrado (TElem => Matrices.TMatriz, "*" => Matrices.Producto_matrices); A: TMatriz := TMatriz.Identidad; **begin** A := Cuadrado_matriz (A); **end** Prueba_operaciones;

Los tipos formales de un genérico se pueden especificar para que pertenezcan a una determinada clase de tipos.

En el cuerpo sólo podemos hacer uso de las propiedades de la clase de tipo del parámetro real. Es decir, a diferencia de las plantillas de C++, la especificación del genérico es un contrato que ha de cumplir la implementación.

38.3 Ver también

- *Polimorfismo* en Wikipedia
- *Generic programming* en Wikipedia en inglés

38.4 Manual de referencia de Ada

- Section 12: Generic Units

Capítulo 39

Programación en Ada/Excepciones

En Ada, cuando se produce algún error durante la ejecución de un programa, se eleva una **excepción**. Dicha excepción puede provocar la terminación abrupta del programa, pero se puede controlar y realizar las acciones pertinentes. También se pueden definir nuevas excepciones que indiquen distintos tipos de error.

39.1 Excepciones predefinidas

En *Ada*, dentro del paquete Standard, existen unas excepciones predefinidas, éstas son:

Constraint_Error cuando se intenta violar una restricción impuesta en una declaración, tal como indexar más allá de los límites de un array o asignar a una variable un valor fuera del rango de su subtipo.

Program_Error se produce cuando se intenta violar la estructura de control, como cuando una función termina sin devolver un valor.

Storage_Error es elevada cuando se requiere más memoria de la disponible.

Tasking_Error cuando hay errores en la comunicación y manejo de tareas.

Numeric_Error en Ada 83 se podía presentar cuando ocurría un error aritmético. A partir del estándar *Ada 95*, desaparece por motivos de portabilidad y pasa a ser un renombrado de Constraint_Error. Por ejemplo, en Ada 83 al dividir entre cero podía saltar Constraint_Error o Numeric_Error (dependiendo del compilador). En Ada 95 este error siempre levanta Constraint_Error.

Name_Error se produce cuando se intenta abrir un fichero que no existe.

39.2 Manejador de excepciones

Cuando se espere que pueda presentarse alguna excepción en parte del código del programa, se puede escribir un manejador de excepciones en las construcciones que lo permitan (bloques o cuerpos de subprogramas, paquetes o tareas), aunque siempre está el recurso de incluir un bloque en cualquier lugar del código.

Su sintaxis sería:

manejador_excepción ::= **when** [identificador :] elección_excepción { | elección_excepción } => secuencia_sentencias elección_excepción ::= identificador | **others**

A la sentencia que comienza por **when**, se le denomina **manejador de excepción**.

La palabra reservada **others** indica cualquier otra excepción y debe ser la única y última opción. Por ejemplo, en un bloque:

begin -- ... **exception when** Constraint_Error => Put (“Error de rango.”); **when** Program_Error | Tasking_Error => Put (“Error de flujo.”); **when others** => Put (“Otro error.”); **end**;

En el momento en el que se produzca la elevación de Constraint_Error durante la ejecución de la secuencia de sentencias entre **begin** y **exception**, el flujo de control se interrumpe y se transfiere a la secuencia de sentencias que siguen a la palabra reservada => del manejador correspondiente.

Otro ejemplo con una función:

function Mañana (Hoy: TDía) **return** TDía **is begin return** TDía'Succ(Hoy); **exception when** Constraint_Error => **return** TDía'First; **end** Mañana;

Nótese que no se puede devolver nunca el control a la unidad donde se elevó la excepción. Cuando se termina la secuencia de sentencias del manejador, termina también la ejecución de dicha unidad.

Si no se controla una excepción, ésta se propaga dinámicamente por las sucesivas unidades invocantes hasta que se maneje en otra o directamente termina la ejecución del programa proporcionando un mensaje con la excepción provocada por pantalla.

39.3 Declaración y elevación de excepciones

Normalmente, es probable prever una situación de error que no se encuentra entre las excepciones predefinidas, por ello, se puede declarar excepciones. Por ejemplo:

Error: **exception**;

Con lo que se puede elevar dicha excepción en el momento pertinente mediante la sentencia raise, cuya sintaxis es:

elevación_excepción ::= **raise** [identificador] ;

Por ejemplo, en un paquete de manejo de una pila estática de números enteros:

package Pila_enteros **is** ErrorPilaEnteros: **exception**; **procedure** Poner (X: Integer); **function** Quitar **return** Integer; **end** Pila_enteros; **package body** Pila_enteros **is** Max: **constant** := 100; Pila: **array** (1..Max) **of** Integer; Cima: Integer **range** 0..Max; **procedure** Poner (X: Integer) **is begin if** Cima = Max **then raise** ErrorPilaEnteros; -- *Se eleva la excepción.* **end if**; Cima := Cima + 1; P(Cima) := X; **end** Poner; **function** Quitar **return** Integer **is begin if** Cima = 0 **then raise** ErrorPilaEnteros; -- *Se eleva la excepción.* **end if**; Cima := Cima - 1; **return** Pila(Cima+1); **end** Quitar; **begin** Cima := 0; **end** Pila_enteros;

Obsérvese que no hace falta else en la sentencias if, pues al elevar la excepción, finaliza la ejecución del subprograma.

Ahora se podría escribir:

declare use Pila_enteros; **begin** Poner (5); -- ... **exception when** ErrorPilaEnteros => -- ... *Manipulación incorrecta de la pila.* **when others** => -- ... **end**;

Si se quiere que dicha excepción no se propague más allá de la unidad en la que se elevó pero no se quiere manejar, se puede emplear una única sentencia vacía (null) dentro de su manejador correspondiente:

procedure Vaciar_pila_enteros **is** Basura: Integer; **use** Pila_enteros; **begin loop** Basura := Quitar; **end loop**; **exception when** ErrorPilaEnteros => **null**; **end** Vaciar_pila_enteros;

Aunque esto no evitaría que se terminara la ejecución de la unidad.

En el caso en el que se quiera propagar una excepción después de haber ejecutado las sentencias pertinentes, se incluiría una sentencia raise dentro del manejador:

-- ... **exception when** ErrorPilaEnteros => Put ("Pila utilizada incorrectamente."); Vaciar_pila_enteros; **raise** ErrorProcesamieto; -- *Se propaga otra excepción.* **end**;

En este caso se propaga otra excepción, pero podría haber sido la misma simplemente con raise, sin crear una nueva ocurrencia de la excepción, por ejemplo:

-- ... **exception when** FalloEnVálvula => Put ("Se ha producido un fallo en la válvula."); **raise**; -- *Se propaga la misma excepción del manejador.* **end**;

Así, se puede realizar un manejo de la excepción en varias capas, realizando sucesivas acciones en cada una de ellas. Dicha sentencia raise sin argumentos debe ser invocada directamente en el manejador, no es posible invocarla en un procedimiento llamado por el manejador.

39.4 Información de la excepción

Ada proporciona información sobre una determinada excepción haciendo uso del paquete predefinido Ada.Exceptions y tras obtener la ocurrencia de la excepción mediante esta notación:

when Ocurrencia : ErrorSensor => Put_Line (Ada.Exceptions.Exception_Information (Ocurrencia));

39.5 Manual de referencia de Ada

- Section 11: Exceptions

39.6 Enlaces externos

- Excepciones: artículo de la Universidad de Valladolid. Ejemplos en Ada y Eiffel.

Capítulo 40

Programación en Ada/Unidades predefinidas/Ada.Exceptions

40.1 Información sobre la excepción

Ada proporciona información sobre una determinada excepción en un objeto del tipo Exception_Ocurrence, definido dentro del paquete predefinido Ada.Exceptions. junto con otras funciones que toman como parámetro una ocurrencia de excepción.

- Exception_Name: devuelve el nombre del tipo de la excepción en notación de puntos completa en letra mayúscula, por ejemplo “PILA.ERROR”.
- Exception_Message: devuelve un mensaje de una línea con la causa y la ubicación de la excepción. No contiene el nombre de la excepción.
- Exception_Information: devuelve una cadena que incluye el nombre de la excepción, la causa y la ubicación y debería proporcionar detalles de la traza de la excepción.

Para comunicar dicha ocurrencia, se emplea la siguiente notación, por ejemplo:

with Ada.Exceptions; **use** Ada.Exceptions; -- ... **exception when** Evento: PresiónElevada | TemperaturaElevada => Put (“Excepción: "); Put (Exception_Name(Evento)); New_Line; **when** Evento: **others** => Put (“Excepción no prevista: "); Put (Exception_Name(Evento)); New_Line; Put (Exception_Message(Evento)); **end**;

En ocasiones puede resultar útil almacenar una ocurrencia. Por ejemplo, sería útil crear un registro de las excepciones producidas en la ejecución de un programa almacenándolas en una lista o vector de ocurrencias. Nótese que el tipo Exception_Occurrence es limitado, de modo que no puede ser asignado para ser guardado. Por esta razón Ada.Exceptions proporciona el procedimiento y la función Save_Ocurrence.

El tipo Exception_Id puede entenderse como un tipo enumeración. Toda excepción declarada mediante la palabra reservada **exception** puede ser considerada como uno de los literales del tipo Exception_Id y, por lo tanto, tiene asociado una identificación, que es el literal correspondiente. Nótese que se refiere ahora a excepciones y a ocurrencias de las mismas. Los literales aludidos serían referencias a Constraint_Error, TemperaturaElevada, etc.; en general, las excepciones predefinidas y las declaradas. Cada una de ellas es en rigor uno de los valores que puede tomar el tipo Exception_Id. El atributo Identity proporciona la identificación de una excepción. Por ejemplo:

declare Fallo_Válvula: **exception**; Id: Ada.Exceptions.Exception_Id; **begin** -- ... Id := Fallo_Válvula'Identity; -- ... **end**;

El programador puede establecer su propio mensaje para una ocurrencia concreta elevándola mediante Raise_Exception en lugar de hacerlo con **raise** seguido del nombre de la excepción. Por ejemplo:

declare Fallo_Válvula : **exception**; **begin** -- ... Raise_Exception (Fallo_Válvula'Identity, “Error de apertura”); -- ... Raise_Exception (Fallo_Válvula'Identity, “Error de cierre”); -- ... **exception when** Evento: Fallo_Válvula => Put(Exception_Message(Evento)); **end**;

Esta llamada es equivalente en Ada 2005 a la siguiente sentencia:

raise Fallo_Válvula **with** “Error de apertura";

40.2 Especificación de Ada.Exceptions

Según el manual de referencia de Ada el paquete Ada.Exceptions debe tener esta especificación:

package Ada.Exceptions **is type** Exception_Id **is private**; Null_Id : **constant** Exception_Id; **function** Exception_Name(Id : Exception_Id) **return** String; **type** Exception_Occurrence **is limited private**; **type** Exception_Occurrence_Access **is access all** Exception_Occurrence; Null_Occurrence : **constant** Exception_Occurrence; **procedure** Raise_Exception(E

: **in** Exception_Id; Message : **in** String := ""); **function** Exception_Message(X : Exception_Occurrence) **return** String; **procedure** Reraise_Occurrence(X : **in** Exception_Occurrence); **function** Exception_Identity(X : Exception_Occurrence) **return** Exception_Id; **function** Exception_Name(X : Exception_Occurrence) **return** String; -- *Same as Exception_Name(Exception_Identity(X)).* **function** Exception_Information(X : Exception_Occurrence) **return** String; **procedure** Save_Occurrence(Target : **out** Exception_Occurrence; Source : **in** Exception_Occurrence); **function** Save_Occurrence(Source : Exception_Occurrence) **return** Exception_Occurrence_Access; **private** ... -- *not specified by the language* **end** Ada.Exceptions;

40.3 Manual de referencia de Ada

- 11.4.1 The Package Exceptions

Capítulo 41

Programación en Ada/Concurrencia

41.1 Concurrencia

La concurrencia es la simultaneidad de hechos. Un programa concurrente es aquel en el que ciertas unidades de ejecución internamente secuenciales (procesos o *threads*), se ejecutan paralela o simultáneamente.

Existen 3 formas básicas de interacción entre procesos concurrentes:

- Sincronización (p.e. las citas o paso de mensajes).
- Señalización (p.e. los semáforos).
- Comunicación (p.e. uso de memoria compartida).

La concurrencia o procesamiento paralelo se ha implementado en leguajes de programación de distinta manera:

- **Programación concurrente clásica**: se basa en la utilización de variables compartidas. Es el caso de *Modula-2* o *Concurrent Pascal*. Para ello, se emplean herramientas como semáforos, regiones críticas y monitores.
- **Programación concurrente distribuida**: se basa en la transferencia de mensajes entre los procesos o *threads*. Es el caso de *C/POSIX*, *Occam* o *Ada*. Se emplean herramientas como canales, buzones y llamadas a procedimiento remoto.

En *Ada* se emplea una programación concurrente distribuida y la principal forma de sincronizar las unidades de ejecución, conocidas como tareas, son los puntos de entrada a la tarea o citas.

41.2 Subsecciones

1. Tareas
2. Sincronización de tareas mediante puntos de entrada o citas (entry)
   (a) Aceptación de citas (accept)
   (b) Selección de citas (select)
   (c) Llamadas a punto de entrada complejas
3. Tareas dinámicas: creación dinámica de tareas (tipos tareas)
   (a) Dependencia de tareas

Capítulo 42

Programación en Ada/Tareas

42.1 Definición de tareas

En *Ada*, a la unidad de proceso secuencial que puede ser ejecutada paralelamente se le denomina *task*. Es la representación explícita de un proceso (o tarea).

Como en otras construcciones, la tarea de *Ada* presenta una especificación (interfaz con el exterior) y un cuerpo (descripción del comportamiento dinámico).

La sintaxis de la especificación de una tarea es:

especificación_tarea ::= **task** identificador [**is** { punto_entrada_tarea | cláusula_representación } [**private** { punto_entrada_tarea | cláusula_representación }] **end** [identificador]] ; punto_entrada_tarea ::= **entry** identificador [(tipo | rango)] [(parámetro { , parámetro })] ;

La sintaxis del cuerpo de una tarea es:

cuerpo_tarea ::= **task body** identificador **is** [parte_declarativa] **begin** secuencia_de_sentencias **end** [identificador] ;

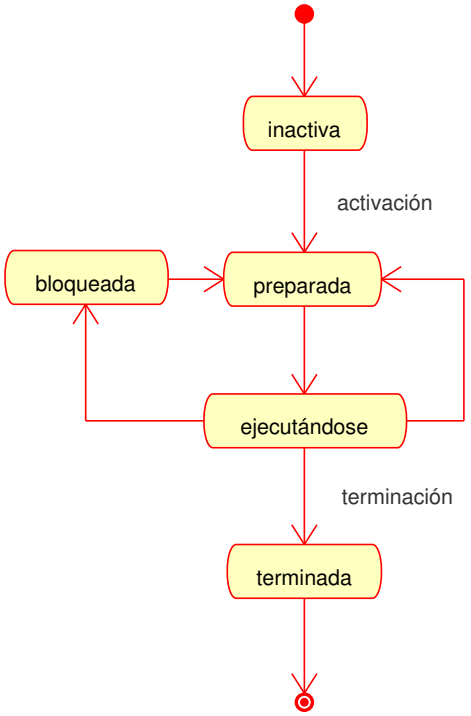
Por ejemplo, dos procesos que escriben un texto por pantalla:

procedure Tareas_tontas **task** Tarea1; **task** Tarea2; **task body** Tarea1 **is begin loop** Put ("Soy la tarea 1."); **end loop**; **end** Tarea1; **task body** Tarea2 **is begin loop** Put ("Soy la tarea 2."); **end loop**; **end** Tarea2; **begin** -- *En este momento comienzan a ejecutarse ambas tareas.* Put ("Soy el procedimiento principal."); **end** Tareas_tontas;

En este caso, el orden de los mensajes que aparecen en pantalla es totalmente impredecible, depende del sistema en el que se ejecuten. Ni siquiera es predecible si las frases que vayan apareciendo serán completas o serán truncadas, esto es debido a que el sistema puede decidir suspender la ejecución de alguna tarea en cualquier instante de tiempo y la sentencia Put no es atómica. Ambas tareas comenzarán su ejecución simultáneamente (al menos lógicamente) justo después del begin del procedimiento.

42.2 Ciclo de vida y tipos

Hay dos tipos de tareas, según su ciclo de vida:



Una tarea Ada puede cambiar de estado cuando se cumplen ciertas condiciones que dependen de si es estática o dinámica.

- **Tareas estáticas**:
  - Dependen del bloque donde se declaran.
  - Se activan justo antes de que se ejecuten las instrucciones del bloque donde fueron declaradas.
  - Terminan cuando todas sus tareas hijas terminen y llegue a su última instrucción.
- **Tareas dinámicas** (mediante punteros):
  - Dependen del bloque donde se define el tipo puntero.
  - Se activan con la sentencia **new**.
  - Las tareas hijas pueden seguir existiendo cuando se termine su progenitora.

Existen atributos asociados a las tareas, entre otros:

- Tarea'Callable indica falso si la tarea está completada o terminada, en otro caso (ejecutándose o suspendida), verdadero.

- Tarea'Terminated indica verdadero si la tarea ha terminado.

Indicar tan sólo, que una tarea se puede terminar abruptamente mediante la sentencia **abort** y para redirigir una llamada a un punto de entrada que esté encolada hacia otra cola, se emplea la sentencia **requeue**.

42.3 Ejemplo

Para un sistema de control de presión y temperatura, se puede realizar el control de ambas magnitudes simultáneamente:

with Paquete_control_temperatura; **use** Paquete_control_temperatura; **with** Paquete_control_presión; **use** Paquete_control_presión; **procedure** Control_temperatura_y_presión **task** Control_temperatura; **task** Control_presión; **task body** Control_temperatura **is** Temperatura: TTemperatura; Temperatura_referencia: TTemperatura := 55; AcciónCalefactor: TAcciónCalefactor; **begin loop** Temperatura := Leer; AccciónCalefactor := Control (Temperatura, Temp_referencia); Escribir (AcciónCalefactor); **delay** (2.0); **end loop**; **end** Control_temperatura; **task body** Control_presión **is** Presión: TPresión; Presión_referencia: TPresión := 315; PosiciónValvula: TPosiciónVálvula; **begin loop** Presión := Leer; PosiciónVálvula := Control (Presión, Presión_referencia); Escribir (PosiciónVálvula); **delay** (3.0); **end loop**; **end** Control_presión; **begin** -- *Comienzan a ejecutarse Control_temperatura y Control_presión;* **null**; -- *Un cuerpo de procedimiento no puede estar vacío.* **end** Control_temperatura_y_presión;

En el preciso instante en el que comienza a ejecutarse el procedimiento, hay tres procesos ejecutándose simultáneamente, el procedimiento y las dos tareas.

42.4 Manual de referencia de Ada

- Section 9: Tasks and Synchronization

Capítulo 43

Programación en Ada/Tareas/Sincronización mediante citas

43.0.1 Sincronización de tareas mediante puntos de entrada o citas (entry)

Frecuentemente, las tareas interaccionan entre sí y necesitan un mecanismo para comunicarse y sincronizarse, este mecanismo que ofrece *Ada* se conoce como la cita (rendezvous) o punto de entrada a la tarea.La cita entre dos tareas se produce como consecuencia de la llamada de una tarea a un punto de entrada declarado en otra tarea.

Los puntos de entrada se declaran en la especificación de la tarea, por ejemplo:

task Tarea **is entry** Entrada (N: Integer); **end** Tarea;

Un punto de entrada se asemeja a un procedimiento. Los parámetros que admiten son de modo in, out o in out, por defecto, se sobreentiende in. Para invocar a un punto de entrada, se procede de igual manera que en un procedimiento, por ejemplo:

T: Tarea; -- ... T.Entrada (8);

Nótese que se debe emplear la notación punto siempre que se realice la llamada fuera de la misma tarea pues una tarea no puede aparecer en una cláusula use. Realizar una llamada a un punto de entrada propio desde la misma tarea está permitido sintácticamente, pero resulta ilógico, pues produciría un interbloqueo consigo misma.

También se pueden definir varios puntos de entrada simultáneamente, por ejemplo:

type TNivel **is** Integer **range** 1..10; **task** Controlador **is** -- *Se define un punto de entrada por cada nivel.* **entry** Aviso (TNivel) (Elem: TElemento); **end** Controlador;

Y se podría llamar a un punto de entrada de los 10 definidos como, por ejemplo:

ProcesoColtrol: Controlador; -- ... ProcesoControl.Aviso (3) (1773);

Con ello, se llama al punto de entrada Aviso con el nivel 3 y parámetro 1173.

Las acciones que se llevan a cabo al aceptar una cita se especifican mediante la sentencia **accept**, tal y como se explica en el apartado siguiente.

43.1 Manual de referencia de Ada

- 9.5.2 Entries and Accept Statements

Capítulo 44

Programación en Ada/Tareas/Aceptación de citas

44.1 Aceptación de citas (accept)

La forma de aceptar una cita y ejecutar las sentencias que se deseen es mediante la sentencia **accept**, dentro del cuerpo de la tarea que acepta la cita. Cada sentencia **entry** debe corresponderse con una sentencia **accept**.

La sintaxis de **accept** es:

acceptación_cita ::= **accept** identificador [(expresión)] [(especificación_parámetro { ; especificación parámetro })] [**do** secuencia_de_sentencias **end** [identificador]] ;

Por ejemplo:

accept Entrada (N: Integer) **do** -- ... *Secuencia de sentencias.* **end** Entrada;

Se deben repetir los parámetros formales declarados en el punto de entrada de la especificación de la tarea.

La diferencias fundamentales entre los puntos de entrada y los procedimientos son:

- El código existente dentro en la sentencia accept es ejecutado por la tarea propietaria y no por la parte invocante, como en los procedimientos.

- Además, hasta que la tarea no llegue a la ejecución de dicha sentencia **accept**, no puede ser invocado el punto de entrada. De igual manera, la parte invocante queda suspendida hasta que termine la ejecución de la sentencia **accept**. Éste es el fundamento de la cita.

La forma más simple de sincronizar una tarea que dependa de la terminación de otro código es por ejemplo:

task Simple **is entry** Continuar; **end** Simple; **task body** Simple **is begin** -- ... **accept** Continuar; -- *Se queda bloqueado hasta que se cite.* -- ... **end** Simple;

Como otro ejemplo, si se quiere implementar una tarea que realice un control de escritura y lectura sobre un buffer de un único elemento:

task Buffer1 **is entry** Escribir (Elem: TElemento); **entry** Leer (Elem: **out** TElemento); **end** Buffer1; **task body** Buffer1 **is** ElemLocal: TElemento; **begin loop accept** Escribir (Elem: TElemento) **do** ElemLocal:= Elem; -- *Guarda el elemento.* **end** Escribir; Ada.Text_IO.Put_Line("Elemento escrito, voy a intentar LEER!"); **accept** Leer (Elem: **out** TElemento) **do** Elem := ElemLocal; -- *Devuelve el elemento.* **end** Escribir; Ada.Text_IO.Put_Line("Elemento leido, vuelvo a intentar ESCRIBIR"); **end loop**; **end** Buffer1;

Se aceptan llamadas Buffer1.Escribir(…) y Buffer1.Leer(…) de forma consecutiva, sin posibilidad de escribir o leer dos o más veces seguidas. Varias tareas diferentes pueden invocar a los puntos de entrada y, por tanto, pueden quedar encoladas. Cada punto de entrada tiene una **cola de tareas** que esperan llamar a dicho punto de entrada. El atributo Escribir'Count contiene el número de tareas que se encuentran encoladas a la espera de que se ejecute el punto de entrada Escribir, pero sólo se puede utilizar dentro de la tarea que contiene el punto de entrada. Con la ejecución de la sentencia **accept** se extraería la primera tarea de la cola (la primera que llegó).

Por tanto, el ejemplo anterior funciona de la siguiente manera: la tarea Buffer1 llega al **accept** de Escribir y se queda bloqueada allí hasta que otra tarea realice una llamada Buffer1.Escribir(…). En ese momento, la tarea Buffer1 ejecuta Escribir y llega al **accept** de Leer, donde se queda bloqueada hasta que otra tarea realice una llamada Buffer1.Leer(…). Se ejecuta Leer y la tarea Buffer1 vuelve al **accept** de Escribir, y así constantemente. Evidentemente, si hay tareas encoladas en los puntos de entrada de Escribir o de Leer, la tarea Buffer1 no se queda bloqueada, sino que atiende a la primera tarea llamante de la cola.

Se puede introducir **código entre dos bloques de accept**, tal y como se ve en el ejemplo anterior: cuando se acaba el primer bloque **accept** (Escribir) se ejecuta dicho código y después se entra en la cola de espera del segundo bloque **accept** (Leer).

Si hay definidos varios puntos de entrada simultáneamente, se puede aceptar uno de ellos, por ejemplo, como:

accept Aviso (3) (Elem: Telemento) **do** -- ... **end** Aviso;

44.2 Manual de referencia de Ada

- 9.5.2 Entries and Accept Statements

Capítulo 45

Programación en Ada/Tareas/Selección de citas

Selección de citas (select)

Uno de los usos de la sentencia select es permitir a una tarea seleccionar entre varias posibles citas; en este caso, se permite su uso únicamente dentro del cuerpo de una tarea. Su sintaxis es la siguiente:

seleccción_aceptación_cita ::= **select** [**when** condición =>] (acceptación_cita | (**delay** [**until**] expresión) [secuencia_de_sentencias]) | (**terminate** ;)) { **or** [**when** condición =>] (acceptación_cita | (**delay** [**until**] expresión) [secuencia_de_sentencias]) | (**terminate** ;)) } [**else** sequencia_sentencias] **end select** ;

Esta alternativa de sentencia **select** permite una combinación de espera y selección entre varias aceptaciones de puntos de entrada a la tarea alternativas. Además, la selección puede depender de condiciones asociadas a cada alternativa.

La sentencia **delay** sirve para indicar que, si en un determinado intervalo de tiempo no se produce ninguna llamada que corresponda con las selecciones anteriores, se ejecuten las sentencias posteriores.

La sentencia **terminate** se elige en la sentencia **select** si la unidad de la que la tarea depende ha llegado al final y todas las tareas hermanas y dependientes han terminado. Es una terminación controlada. Esta alternativa no puede aparecer si hay una alternativa **delay** o **else**.

Por ejemplo:

task Servidor **is entry** Trabajar; **entry** Cerrar; **end**; **task body** Servidor **is begin loop select accept** Trabajar **do** -- *Se acepta la llamada a trabajar.* Trabajando := True; -- *Variable global.* **end**; Trabajo_servidor; -- *Trabaja.* **or accept** Cerrar; -- *Se cierra el servidor.* **exit**; **or delay** (60.0); -- *¿Se han olvidado del servidor?* Put ("Estoy esperando trabajar."); -- *Otra opción en vez de delay:* -- *or* -- --*Terminación normal cuando se destruya el objeto tarea.* -- *terminate;* **end select**; **end loop**; **end** Servidor;

Como otro ejemplo, para garantizar la exclusión mutua a una variable (acceso seguro a memoria compartida), se podría implementar con tareas de la siguiente manera:

task Variable_protegida **is entry** Leer (Elem: **out** TElemento); **entry** Escribir (Elem: TElemento); **end**; **task body** Variable_protegida **is** ElemLocal: TElemento; **begin accept** Escribir (Elem: TElemento) **do** ElemLocal := Elem; **end** Escribir; **loop select accept** Escribir (Elem: TElemento) **do** ElemLocal := Elem; **end** Escribir; **or accept** Leer (Elem: **out** TElemento) **do** Elem := ElemLocal; **end** Leer; **end select**; **end loop**; **end** Variable_protegida;

La primera sentencia de la tarea es un **accept** de Escribir, con lo que se asegura que la primera llamada le de un valor a la variable local. En el supuesto de que se realizara una llamada a Leer, ésta quedaría encolada hasta que se produjera la aceptación de Escribir. Después, la tarea entra en el bucle infinito que contiene una sentencia **select**. Es ahí donde se acepta tanto llamadas a Escribir como a Leer de la siguiente manera:

Si no se llama ni a Leer ni a Escribir, entonces la tarea se queda suspendida hasta que se llame a algún punto de entrada, en ese momento se ejecutará la sentencia **accept** correspondiente.

Si hay una o más llamadas en la cola de Leer, pero no hay llamadas en la de Escribir, se acepta la primera llamada a Leer, y viceversa.

Si hay llamadas tanto en la cola de Leer como en la de Escribir, se hace una elección arbitraria.

Es una tarea que sirve a dos colas de clientes que esperan servicios diferentes. Sin embargo, se impide el acceso múltiple a la variable local.

45.1 Manual de referencia de Ada

- 9.7 Select Statements

Capítulo 46

Programación en Ada/Tareas/Llamadas a punto de entrada complejas

46.1 Llamadas a punto de entrada complejas

A veces, interesa que una llamada a un punto de entrada de una tarea cumpla unos requisitos. Esto es debido a que se puede bloquear el proceso que realiza la llamada y puede ser interesante disponer de métodos para desbloquearlo si no se cumplen unas determinadas condiciones.

La sentencia select, además de servir como selección de aceptaciones de puntos de entrada dentro del cuerpo de la tarea que los contiene, también proporciona mecanismos para seleccionar el comportamiento de las llamadas a puntos de entrada. Su sintaxis es la siguiente:

llamada_a_punto_de_entrada_compleja ::= llamada_a_punto_de_entrada_con_tiempo_límite | llamada_a_punto_de_entrada_condicional | llamada_a_punto_de_entrada_asíncrona llamada_a_punto_de_entrada_con_tiempo_límite ::= **select** identif_p_entrada [(tipo | rango)] [(parámetro { , parámetro })] ; [secuencia_de_sentencias] **or delay** [**until**] expresión ; [secuencia_de_sentencias] **end select** ; llamada_a_punto_de_entrada_condicional ::= **select** identif_p_entrada [(tipo | rango)] [(parámetro { , parámetro })] ; [secuencia_de_sentencias] **else** secuencia_de_sentencias **end select** ; llamada_a_punto_de_entrada_asíncrona ::= **select** (identif_p_entrada [(tipo | rango)] [(parámetro { , parámetro })] ;) | (**delay** [**until**] expresión ;) [secuencia_de_sentencias] **then abort** secuencia_de_sentencias **end select** ;

46.2 Tipos de punto de entrada

Como puede apreciarse, hay tres posibles llamadas a puntos de entrada a parte de la simple, éstas son: llamada con tiempo límite, llamada condicional y transferencia asíncrona.

46.2.1 Llamada con tiempo límite

Llama a un punto de entrada que es cancelado si no se produce la aceptación antes de que finalice un plazo de tiempo. Ejemplo:

select Controlador.Petición (Medio) (Elem); **or delay** 50.0; Put ("Controlador demasiado ocupado."); **end select**;

46.2.2 Llamada condicional

Llama a un punto de entrada que es cancelada si no es aceptada inmediatamente, es decir, tiene un tiempo límite nulo. Ejemplo:

select Procesado.Aviso; **else raise** Error; **end select**;

46.2.3 Transferencia asíncrona

Proporciona la transferencia asíncrona de control cuando se acepte la llamada a un punto de entrada o se cumpla un plazo de tiempo, mientras se esté ejecutando una secuencia de sentencias. Es decir, si se acepta la llamada al punto de entrada o cumple el plazo, se abortan las sentencias que se estuvieran ejecutando. Ejemplo:

select delay 5.0; **raise** FunciónNoConverge; **then abort** Función_recursiva (X, Y); **end select**;

46.3 Manual de referencia de Ada

- 9.7.2 Timed Entry Calls
- 9.7.3 Conditional Entry Calls
- 9.7.4 Asynchronous Transfer of Control

Capítulo 47

Programación en Ada/Tareas/Dinámicas

47.1 Creación dinámica de tareas (tipos tareas)

Además de poder declarar tareas como un simple objeto, se pueden declarar un tipo como tipo tarea. Con ello se consigue poder utilizar otros objetos que utilicen dicho tipo tarea como por ejemplo, un vector de tareas o un puntero a tarea (con lo que se consigue crear tareas dinámicamente).

La sintaxis de los tipos tarea es la siguiente:

declaración_tipo_tarea ::= **task type** identificador [(discriminante { ; discriminante })] [**is** { punto_entrada_tarea | cláusula_representación } [**private** { punto_entrada_tarea | cláusula_representación }] **end** [identificador]] ; discriminante ::= identificador { , identificador } : [**access**] subtipo [:= expresión]

Con ello, se define un nuevo tipo. Esta definición necesita una terminación, es decir, faltaría declarar el cuerpo de la tarea, que se realiza de igual manera que si se hubiera declarado la tarea simplemente.

Los tipos tarea son privados limitados. Es decir, un objeto declarado de tipo tarea no es una variable, se comporta como una constante. Por tanto, no se permite la asignación, igualdad y desigualdad para los tipos tarea.

Según la sintaxis descrita, se puede definir una tarea como un tipo, por ejemplo, de esta manera:

declare task type TTareaA; **task type** TTareaB; **task type** TTareasMúltiples; **type** TVectorTareas **is array** (1..10) **of** TTareasMúltiples; A: TTareaA; B: TTareaB; V: TVectorTareas; **task body** TTareaA **is** -- ... **end**; **task body** TTareaB **is** -- ... **end**; **task body** TTareasMúltiples **is** -- ... **end**; **begin** -- *A partir de aquí se ejecutan las 12 tareas concurrentemente.* -- ... **end**;

En el momento en el que dé comienzo la ejecución del bloque, justo después de **begin**, dará comienzo la ejecución simultánea de las tareas definidas en las distintas variables del tipo **task**. En este caso TTareaA, TTareaB y 10 TTareasMúltiples. Pero esta situación es estática, no se pueden lanzar tareas en un determinado instante; para ello, se pueden emplear punteros como, por ejemplo:

procedure Ejemplo_tareas_dinámicas **is task type** TTarea; **type** PTTarea **is access** TTarea; T1: PTTarea; T2: PTTarea := **new** TTarea; -- *Se crea la tarea T2.all.* **begin** T1 := **new** TTarea; -- *Se crea la tarea T1.all.* T1 := **null**; -- *Se pierde la referencia, pero se sigue ejecutando.* -- ... **end** Ejemplo_tareas_dinámicas;

Las tareas creadas con **new** siguen unas reglas de activación y dependencia ligeramente diferentes. Estas tareas inician su activación inmediatamente después de la evaluación del asignador de la sentencia **new**. Además, estas tareas no dependen de la unidad donde se crearon, sino que dependen del bloque, cuerpo de subprograma o cuerpo de tarea que contenga la declaración del tipo **access** en sí. Para referenciar a tareas dinámicas se emplea el nombre de la variable puntero seguido de .**all**, por ejemplo, T1.**all**. Si se quiere que termine una tarea creada dinámicamente se debe utilizar la sentencia **abort**. Por ejemplo, **abort** T1.**all**.

También se pueden crear varios ejemplares de un mismo tipo dependiendo de un parámetro denominado discriminante. Por ejemplo:

task type TManejadorTeclado (ID: TIDentifTeclado := IDPorDefecto) **is entry** Leer (C: **out** Character); **entry** Escribir (C: **in** Character); **end** TManejadorTeclado; **type** PTManejadorTeclado **is access** TManejadorTeclado; Terminal: PTManejadorTeclado := **new** TManejadorTeclado (104);

47.2 Manual de referencia de Ada

- 9.1 Task Units and Task Objects

Capítulo 48

Programación en Ada/Tareas/Dependencia

48.1 Dependencia de tareas

Las reglas de dependencia de las tareas son:

- Si la tarea es creada por la elaboración de una declaración de objeto, depende de la unidad que incluya dicha elaboración.
- Si la tarea es creada por la evaluación de una sentencia **new** para un tipo puntero dado, depende de cada unidad que incluya la elaboración de la declaración de dicho tipo puntero.

Por ejemplo:

declare task type TTarea; **type** PTTareaGlobal **is access** TTarea; T1, T2: TTarea; PunteroTareaGlobal1: PTTareaGlobal; **begin** -- *Se activan T1 y T2.* **declare type** PTTareaLocal **is access** TTarea; PunteroTareaGlobal2: PTTareaGlobal := **new** TTarea; -- *Se activa PunteroTareaGlobal2.all después de la asignación new.* PunteroTareaLocal: PTTareaLocal := **new** TTarea; -- *Se activa PunteroTareaLocal.all después de la asignación new.* T3: TTarea; **begin** -- *Se activa T3.* -- ... **end**; -- *Se espera la terminación de T3 y PunteroTareaLocal.all.* -- *Continúa la ejecución de PunteroTareaGlobal2.all.* -- ... **end**; -- *Se espera la terminación de T1, T2, PunteroTareaGlobal1.all* -- *y PunteroTareaGlobal2.all.*

48.2 Manual de referencia de Ada

- 9.3 Task Dependence - Termination of Tasks

Capítulo 49

Programación en Ada/Tareas/Ejemplos

49.1 Ejemplos completos de tareas

49.1.1 Semáforos

Una posible implementación del tipo abstracto semáforo es con tareas Ada. Pero este ejemplo no se ha de tomar muy en serio, puesto que es un típico caso de inversión de la abstracción, es decir, se hace uso de un mecanismo de alto nivel, las tareas, para implementar uno de bajo nivel, los semáforos. En Ada 95 la mejor manera de implementar un semáforo es un objeto protegido. Sin embargo a efectos didácticos es un buen ejemplo.

generic ValorInicial: Natural := 1; -- Parám. genérico con valor por defecto. package Semaforos is type TSemaforo is limited private; procedure Wait (Sem: in out TSemaforo); procedure Signal (Sem: in out TSemaforo); private task type TSemaforo is entry Wait; entry Signal; end TSemaforo; end Semaforos;
package body Semaforos is procedure Wait (Sem: in out TSemaforo) is begin Sem.Wait; -- Llamada a punto de entrada de la tarea. end Wait; procedure Signal (Sem: in out TSemaforo) is begin Sem.Signal; -- Llamada a punto de entrada de la tarea. end Signal; task body TSemaforo is S: Natural := ValorInicial; -- Es el contador del semáforo. begin loop select when S > 0 => accept Wait; S := S - 1; or accept Signal; S := S + 1; or terminate; end select; end loop; end TSemaforo; end Semaforos;
with Semaforos; procedure Prueba_Semaforos is package Paquete_Semaforos is new Semaforos; use Paquete_Semaforos; Semaforo: TSemaforo; begin -- Aquí se inicia la tarea de tipo TSemaforo (objeto Semaforo). -- ... Wait (Semaforo); -- ... Signal (Semaforo); -- ... end Prueba_Semaforos;

49.1.2 Simulación de trenes

Problema: **Escribe un programa que realice una simulación de trenes circulando por estaciones. Cada tren espera a que la estación siguiente esté libre para avanzar, es decir, hasta que un tren no ha abandonado una estación, el tren de la estación anterior no puede avanzar. Para ello puedes usar los semáforos**

ejercicio

definidos en el ejemplo anterior.

Solución:

Solución propuesta:

with Ada.Text_IO; use Ada.Text_IO; with Ada.Numerics.Float_Random; with Semaforos; procedure Simulador_Trenes is Num_Estaciones : constant := 5; Num_Trenes : constant := 3; type Num_Estación is range 1 .. Num_Estaciones; type Num_Tren is range 1 .. Num_Trenes; package Num_Estación_IO is new Ada.Text_IO.Integer_IO (Num_Estación); use Num_Estación_IO; package Num_Tren_IO is new Ada.Text_IO.Integer_IO (Num_Tren); use Num_Tren_IO; package Semaforos_Inicial_1 is new Semaforos (Valorinicial => 1); use Semaforos_Inicial_1; Semaforos_Estaciones : array (Num_Estación) of TSemaforo; task type Tren is entry Comenzar (Tu_Num : in Num_Tren); end Tren; Lista_Trenes : array (Num_Tren) of Tren; task body Tren is Mi_Num: Num_Tren; procedure Pon_Nombre is begin Put ("Tren nº"); Put (Mi_Num); Put (": "); end Pon_Nombre; Espera_En_Estación: constant Du-

ration := 5.0; Duración_Mínima: constant Duration := 2.0; Factor_Duración: constant Duration := 10.0; Azar_Gen: Ada.Numerics.Float_Random.Generator; Actual, Siguiente: Num_Estación; begin Ada.Numerics.Float_Random.Reset (Azar_Gen); accept Comenzar (Tu_Num : in Num_Tren) do Mi_Num := Tu_Num; end Comenzar; Pon_Nombre; Put_Line ("Comienzo el trayecto"); Actual := 1; loop Pon_Nombre; Put ("En estación "); Put (Actual); New_Line; delay Espera_En_Estación; if Actual = Num_Estaciones then Siguiente := 1; else Siguiente := Actual + 1; end if; Wait (Semaforos_Estaciones (Siguiente)); Pon_Nombre; Put ("Trayecto hacia estación "); Put (Siguiente); New_Line; Signal (Semaforos_Estaciones (Actual)); delay Duration (Ada.Numerics.Float_Random.Random (Azar_Gen)) * Factor_Duración + Duración_Mínima; Actual := Siguiente; end loop; end Tren; begin for I in Lista_Trenes'Range loop Lista_Trenes (I).Comenzar (Tu_Num => I); end loop; end Simulador_Trenes;

49.1.3 *Buffer* circular

Otro ejemplo, una posible implementación de un *buffer* circular:

generic type TElemento is private; Tamaño: Positive := 32; package Buffer_servidor is type TBuffer is limited private; procedure EscribirBuf (B: in out TBuffer; E: TElemento); procedure LeerBuf (B: in out TBuffer; E: out TElemento); private task type TBuffer is entry Escribir (E: TElemento); entry Leer (E: out TElemento); end TBuffer; end Buffer_servidor;
package body Buffer_servidor is task body TBuffer is subtype TCardinalBuffer is Natural range 0 .. Tamaño; subtype TRangoBuffer is TCardinalBuffer range 0 .. Tamaño - 1; Buf: array (TRangoBuffer) of TElemento; Cima, Base: TRangoBuffer := 0; NumElementos: TCardinalBuffer := 0; begin loop select when NumElementos < Tamaño => accept Escribir (E: TElemento) do Buf(Cima) := E; end Escribir; Cima := TRangoBuffer(Integer(Cima + 1) mod Tamaño); NumElementos := NumElementos + 1; or when NumElementos > 0 => accept Leer (E: out TElemento) do E := Buf(Base); end Leer; Base := TRangoBuffer(Integer(Base + 1) mod Tamaño); NumElementos := NumElementos - 1; or terminate; end select; end loop; end TBuffer; procedure EscribirBuf (B: in out TBuffer; E: TElemento) is begin B.Escribir (E); end EscribirBuf; procedure LeerBuf (B: in out TBuffer; E: out TElemento) is begin B.Leer (E); end LeerBuf; end Buffer_servidor;
with Text_IO, Buffer_servidor; use Text_IO; procedure Buffer is Clave_Salida : constant String := "Salir"; type TMensaje is record NumOrden: Positive; Contenido: String (1..20); end record; package Cola_mensajes is new Buffer_servidor (TElemento => TMensaje); use Cola_mensajes; Cola: TBuffer; task Emisor; task Receptor; task body Emisor is M: TMensaje := (NumOrden => 1, Contenido => (others => ' ')); Último: Natural; begin loop Put ("[Emisor] Mensaje: "); Get_Line (M.Contenido, Último); M.Contenido (Último + 1 .. M.Contenido'Last) := (others => ' '); EscribirBuf (Cola, M); M.NumOrden := M.NumOrden + 1; exit when M.Contenido(Clave_Salida'range) = Clave_Salida; end loop; end Emisor; task body Receptor is package Ent_IO is new Text_IO.Integer_IO(Integer); use Ent_IO; M: TMensaje; begin loop LeerBuf (Cola, M); exit when M.Contenido(Clave_Salida'range) = Clave_Salida; Put ("[Receptor] Mensaje número "); Put (M.NumOrden); Put (": "); Put (M.Contenido); New_Line; end loop; end Receptor; begin null; end Buffer;

49.1.4 Problema del barbero durmiente

Esta es una solución al problema del barbero durmiente.

with Ada.Text_IO; use Ada.Text_IO; with Ada.Numerics.Discrete_Random; procedure Barberia is type Rango_Demora is range 1 .. 30; type Duracion_Afeitado is range 5 .. 10; type Nombre_Cliente is (Jose, Juan, Iñaki, Antonio, Camilo); package Demora_Al_Azar is new Ada.Numerics.Discrete_Random (Rango_Demora); package Afeitado_Al_Azar is new Ada.Numerics.Discrete_Random (Duracion_Afeitado); task Barbero is entry Afeitar (Cliente : in Nombre_Cliente); end Barbero; task type Cliente is entry Comenzar (Nombre : in Nombre_Cliente); end Cliente; Lista_Clientes : array (Nombre_Cliente) of Cliente; task body Barbero is Generador : Afeitado_Al_Azar.Generator; Espera_Máxima_Por_Cliente : constant Duration := 30.0; begin Afeitado_Al_Azar.Reset (Generador); Put_Line ("Barbero: Abro la barbería."); loop Put_Line ("Barbero: Miro si hay cliente."); select accept Afeitar (Cliente : in Nombre_Cliente) do Put_Line ("Barbero: Afeitando a " & Nombre_Cliente'Image (Cliente)); delay Duration (Afeitado_Al_Azar.Random (Generador)); Put_Line ("Barbero: Termino con " & Nombre_Cliente'Image (Cliente)); end Afeitar; or delay Espera_Máxima_Por_Cliente; Put_Line ("Barbero: Parece que ya no viene nadie," & " cierro la barbería."); exit; end select; end loop; end Barbero; task body Cliente is Generador : Demora_Al_Azar.Generator; Mi_Nombre : Nombre_Cliente; begin accept Comenzar (Nombre : in Nombre_Cliente) do Mi_Nombre := Nombre; end Comenzar; Demora_Al_Azar.Reset (Gen => Generador, Initiator => Nombre_Cliente'Pos (Mi_Nombre)); delay Duration (Demora_Al_Azar.Random (Generador)); Put_Line (Nombre_Cliente'Image (Mi_Nombre) & ": Entro en la barbería."); Barbero.Afeitar (Cliente => Mi_Nombre); Put_Line (Nombre_Cliente'Image (Mi_Nombre) & ": Estoy afeitado, me marcho."); end Cliente; begin for I in Lista_Clientes'Range loop Lista_Clientes (I).Comenzar (Nombre => I); end loop; end Barberia;

49.1.5 Problema de los filósofos cenando

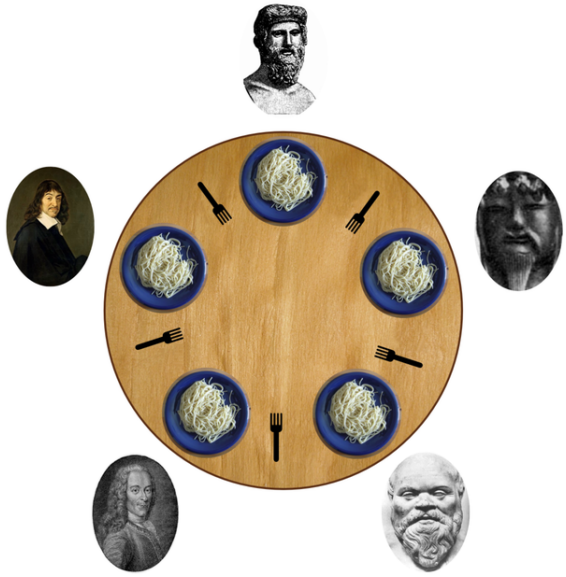


Ilustración del problema de los filósofos cenando

Una solución con tareas y objetos protegidos del conocido problema de los filósofos cenando.

```
package Cubiertos is type Cubierto is limited private; procedure Coger(C: in out Cubierto); procedure Soltar(C: in out Cubierto); private type Status is (LIBRE, OCUPADO); protected type Cubierto(Estado_Cubierto: Status := LIBRE) is entry Coger; entry Soltar; private Estado: Status := Estado_Cubierto; end Cubierto; end Cubiertos;
package body Cubiertos is procedure Coger (C: in out Cubierto) is begin C.Coger; end Coger; procedure Soltar (C: in out Cubierto) is begin C.Soltar; end Soltar; protected body Cubierto is entry Coger when Estado = LIBRE is begin Estado := OCUPADO; end Coger; entry Soltar when Estado = OCUPADO is begin Estado := LIBRE; end Soltar; end Cubierto; end Cubiertos;
with Ada.Text_IO; use Ada.Text_IO; with Ada.Integer_Text_IO; use Ada.Integer_Text_IO; with Cubiertos; use Cubiertos; procedure Problema_Filosofos is type PCubierto is access Cubierto; task type TFilosofo(Id: Character; Cubierto1: PCubierto; Cubierto2: PCubierto); task body TFilosofo is procedure Comer is begin Coger(Cubierto1.all); Coger(Cubierto2.all); for i in 1..10 loop Put(Id & “c "); delay 1.0; end loop; Soltar(Cubierto2.all); Soltar(Cubierto1.all); end Comer; Procedure Pensar is begin for i in 1..10 loop Put(Id & “p "); delay 1.0; end loop; end Pensar; begin loop Comer; Pensar; end loop; end TFilosofo; Num_Cubiertos: Positive; begin Put(“Introduce el numero de cubiertos: "); Get(Num_Cubiertos); New_line; declare type PTFilosofo is access TFilosofo; P: PTFilosofo; C: Character := 'A'; Cuberteria: array (1..Num_Cubiertos) of PCubierto; begin for i in 1..Num_Cubiertos loop Cuberteria(i) := new Cubierto; end loop; for i in 1..Num_Cubiertos-1 loop P := new TFilosofo(C, Cuberteria(i), Cuberteria(i+1)); C := Character'Succ(C); end loop; P := new TFilosofo(C, Cuberteria(1), Cuberteria(Num_Cubiertos)); end; end Problema_Filosofos;
```

Para evitar el bloqueo mutuo es totalmente imprescindible que al último filósofo se le asignen los cubiertos en ese orden. Sí se hiciese al contrario, el bloqueo no tardaría en aparecer (sobre todo si se eliminan las instrucciones *delay*):

```
P := new TFilosofo(C, Cuberteria(Num_Cubiertos), Cuberteria(1));
```

49.1.6 Chinos: una implementación concurrente en Ada

```
-- Chinos2: Otra implementación concurrente en Ada -- Tomás Javier Robles Prado -- tjavier@usuarios.retecal.es -- Uso: ./chinos <numero_jugadores> -- El juego consiste en jugar sucesivas partidas a los chinos. Si un -- jugador acierta, no paga y queda excluido de las siguientes -- rondas. El último que quede paga los vinos -- Copyright (C) 2003 T. Javier Robles Prado -- -- This program is free software; you can redistribute it and/or modify -- it under the terms of the GNU General Public License as published by -- the Free Software Foundation; either version 2 of the License, or -- (at your option) any later version. -- -- This program is distributed in the hope that it will be useful, -- but WITHOUT ANY WARRANTY; without even the implied warranty of -- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the -- GNU General Public License for more details. -- -- You should have received a copy of the GNU General Public License -- along with this program; if not, write to the Free Software -- Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA with Ada.Text_IO; use Ada.Text_IO; with Ada.Numerics.Discrete_Random; with Ada.Command_Line; with Ada.Strings.Unbounded; with Ada.Exceptions; procedure Chinos is -- Número Máximo Jugadores que pueden participar (aforo máximo del bar) MAX : constant Natural := 20; -- Posibles mensajes que recibe un jugador tras una partida type Estados is (NO_SIGUES_JUGANDO, SIGUES_JUGANDO, HAS_PERDIDO); -- Subtipo que modela el número de jugadores posibles subtype NumMaxJugadores is Natural range 0..MAX; -- Modela la máxima apuesta que puede darse subtype MAX_APUESTA is Natural range 0..3*MAX; -- Nombres posibles para los jugadores. El 0 se utilizará para -- controlar el caso de que no haya ganador en una partida subtype TNombre is Integer range −1..MAX; -- Paquete para Numeros aleatorios: package Integer_Random is new
```

```
Ada.Numerics.Discrete_Random(MAX_APUESTA); -- Apuesta de cada Jugador Subtype TApuesta is Integer range −1..3*MAX; -- Mano de cada jugador subtype TMano is Natural range 0..3; -- Ficha de cada jugador que guardara el arbitro type TFicha is record Nombre : TNombre; Apuesta : TApuesta := −1; Mano : TMano; SigueJugando : Boolean; end record; -- Array de Fichas type TTablon is array(1..MAX) of TFicha; -- Se define el tipo jugador task type Jugador; task Arbitro is -- El árbitro controla las partidas y sincroniza a los jugadores entry FijaNumeroJugadores (Num : in NumMaxJugadores); -- Recoge el argumento de la línea de comandos para saber -- cuántos jugadores van a participar entry AsignaNombre (Nombre: out TNombre; NumJug: out NumMaxJugadores); -- Asigna Nombres (de 1 a NumerosJugadores) a los jugadores que -- van a participar. A los que no, les asigna un −1 como -- indicación de que finalicen. entry SiguesJugando (Nombre: in TNombre; JugadorSigueJugando : out Estados; HuboGanador : out boolean); -- Mensaje que envía el árbitro a cada jugador tras una -- partida, comunicándole si ha ganado y deja de jugar, si -- sigue jugando o si ha perdido y tiene que pagar entry EnviaApuesta (Nombre: in TNombre ; Apuesta: in TApuesta); -- El árbitro recibe la apuesta de un jugador entry ConfirmaApuesta (Confirmada : out Boolean); -- Respuesta del árbitro sobre si la apuesta es válida (no la -- ha hecho otro antes) entry ReEnviaApuesta (Apuesta: in TApuesta); -- Si la apuesta no es válida se reenvia hasta que lo sea entry EnviaMano (Nombre: in TNombre ; Mano: in TMano); -- El jugador envía el número de manos que saca al árbitro end Arbitro; task body Arbitro is -- Funciones y Procedimientos function NumeroJugadores return NumMaxJugadores is -- Devuelve el número de jugadores begin return 5; end NumeroJugadores; function EsApuestaValida (Apuesta: in TApuesta; Tablon: in TTablon) return Boolean is -- Devuelve verdadero si la apuesta no ha sido realizada -- antes por algún otro jugador Valida : Boolean := True ; I : TNombre := 1; begin for I in 1..MAX loop if Tablon(I).SigueJugando then if Tablon(I).Apuesta = Apuesta then -- Ya está dicha, la apuesta NO es válida Valida := False ; end if; end if; end loop; return Valida; end EsApuestaValida; function ResultadoGanador (Tablon: in TTablon) return TApuesta is -- Devuelve el número de monedas que sacaron los jugadores Suma : TApuesta := 0 ; begin for I in 1..MAX loop if Tablon(I).SigueJugando then Suma := Suma + Tablon(I).Mano ; end if; end loop; return Suma; end ResultadoGanador; procedure ImprimeGanador (Tablon: in TTablon) is -- Imprimer el nombre del ganador I : TNombre := 1 ; Resultado : TApuesta ; Terminar : Boolean := False; begin Resultado := ResultadoGanador(Tablon); while not Terminar loop if Tablon(I).Apuesta = Resultado and Tablon(I).SigueJugando then Put_Line("Ha Ganado el Jugador " & I'Img); Terminar := True ; else if I = MAX then Put_Line("No ha habido Ganador"); Terminar := True; else I := I + 1; end if; end if; end loop; end ImprimeGanador; function JugadorEliminado (Tablon: in TTablon) return NumMaxJugadores is -- Devuelve el jugador que cuya apuesta sea la correcta Resultado : TApuesta; Ganador : NumMaxJugadores := 0; begin Resultado := ResultadoGanador(Tablon); for I in 1..MAX loop if Tablon(I).SigueJugando then if Resultado = Tablon(I).Apuesta then Ganador := I ; end if; end if; end loop; return Ganador; end JugadorEliminado; procedure ImprimeTablon(Tablon: in TTablon) is -- Imprime las apuestas y monedas de los jugadores begin for I in 1..MAX loop if Tablon(I).SigueJugando then Put_Line("Nombre =" & Tablon(I).Nombre'Img & " | Apuesta =" & Tablon(I).Apuesta'Img & " | Mano =" &Tablon(I).Mano'Img ); end if; end loop; Put_Line ("Resultado ganador: " & ResultadoGanador(Tablon)'Img); end ImprimeTablon; procedure SeparaPartidas (NumPar :in Natural) is -- Un simple separador para aumentar la claridad begin New_Line; Put_Line("****************************************"); Put_Line("Partida número " & NumPar'Img); Put_Line("****************************************"); end SeparaPartidas; -- Variables -- Número de jugadores de la partida N : NumMaxJugadores; Permitidos : NumMaxJugadores; -- Partida Actual PartidaActual : NumMaxJugadores; -- Tablón Tablon : TTablon; NombreActual : NumMaxJugadores; ApuestaValida : Boolean; Ganador : NumMaxJugadores; NumeroPartida : Natural; begin -- Averigua número de jugadores accept FijaNumeroJugadores (Num : in NumMaxJugadores) do N := Num; end FijaNumeroJugadores; -- Nombra solo a aquellos que vayan a jugar, a los que no, los -- nombra como −1 Permitidos := N; for I in 1..MAX loop accept AsignaNombre (Nombre: out TNombre ; NumJug: out NumMaxJugadores) do if Permitidos > 0 then Nombre := I; NumJug := N; Tablon(I).Nombre := I ; Tablon(I).SigueJugando := True; Permitidos := Permitidos - 1; else Nombre := −1; Tablon(I).Nombre := −1; Tablon(I).SigueJugando := False; end if; end AsignaNombre; end loop; NumeroPartida := 1; while N /= 1 loop -- Para separar las diferentes partidas SeparaPartidas(NumeroPartida); -- Recibe las apuestas de cada jugador for I in 1..N loop accept EnviaApuesta (Nombre: in TNombre; Apuesta: in TApuesta) do NombreActual := Nombre; ApuestaValida := EsApuestaValida(Apuesta,Tablon); if ApuestaValida then Tablon(Nombre).Apuesta := Apuesta ; end if; end EnviaApuesta; -- La Apuesta es Válida, se confirma y a otra cosa if ApuestaValida then accept ConfirmaApuesta(Confirmada: out Boolean) do Confirmada := True; end ConfirmaApuesta; else -- La apuesta no es válida. Se comunica esto al jugador para -- que envíe una nueva apuesta accept ConfirmaApuesta(Confirmada: out Boolean) do Confirmada := False; end ConfirmaApuesta; while not ApuestaValida loop -- Aceptará diferentes apuestas hasta q sea válida. accept ReEnviaApuesta (Apuesta: in TApuesta) do if EsApuestaValida(Apuesta,Tablon) then ApuestaValida := True; Tablon(NombreActual).Apuesta := Apuesta
```

; end if; end ReEnviaApuesta; accept ConfirmaApuesta(Confirmada: out Boolean) do Confirmada := ApuestaValida; end ConfirmaApuesta; end loop; end if; end loop; -- Recibe lo q saca cada jugador for I in 1..N loop accept EnviaMano(Nombre: in TNombre; Mano: in TMano) do Tablon(Nombre).Mano := Mano ; end EnviaMano; end loop; -- ImprimeResultados de la partida ImprimeTablon(Tablon); ImprimeGanador(Tablon); -- Envía a cada jugador su nuevo estado Ganador := JugadorEliminado (Tablon); if Ganador = 0 then -- Nadie acertó for I in 1..N loop accept SiguesJugando (Nombre: in TNombre; JugadorSigueJugando : out Estados; HuboGanador : out boolean) do JugadorSigueJugando := SIGUES_JUGANDO; Tablon(Nombre).SigueJugando := True; HuboGanador := false ; end SiguesJugando; end loop; else -- Hay ganador for I in 1..N loop accept SiguesJugando (Nombre: in TNombre; JugadorSigueJugando : out Estados; HuboGanador : out boolean) do HuboGanador := true; if Nombre = Ganador then JugadorSigueJugando := NO_SIGUES_JUGANDO; Tablon(Nombre).SigueJugando := False; else if N /= 2 then JugadorSigueJugando := SIGUES_JUGANDO; Tablon(Nombre).SigueJugando := True; else JugadorSigueJugando := HAS_PERDIDO; Tablon(Nombre).SigueJugando := False; end if; end if; end SiguesJugando; end loop; end if; NumeroPartida := NumeroPartida + 1; if Ganador /= 0 then N := N - 1; end if; end loop; end Arbitro; task body Jugador is MiNombre : TNombre; NumJug : NumMaxJugadores; Apuesta : TApuesta; ApuestaValidada : Boolean; Mano : Tmano; G : Integer_Random.Generator; YoSigo : Estados; Terminar : Boolean := False; HuboGanador : boolean; begin Arbitro.AsignaNombre(MiNombre, NumJug); -- Si MiNombre es −1, entonces termina su ejecución. Se sigue -- este método para ceñirnos a los jugadores que quiere el -- usuario if MiNombre /= −1 then -- Semillas aleatorias Integer_Random.Reset(G); while not Terminar loop -- Envia Apuesta for I in 1..MiNombre loop Apuesta := Integer_Random.Random(G) mod (NumJug * 3); end loop; Arbitro.EnviaApuesta(MiNombre, Apuesta); -- Proceso de confirmación de apuesta ApuestaValidada := False ; while not ApuestaValidada loop Arbitro.ConfirmaApuesta(ApuestaValidada); if not ApuestaValidada then -- Genera Nueva apuesta for I in 1..MiNombre loop Apuesta := Integer_Random.Random(G) mod (NumJug * 3) ; end loop; Arbitro.ReEnviaApuesta(Apuesta); end if; end loop; -- Envía Mano for I in 1..MiNombre loop Mano := Integer_Random.Random(G) mod 4; end loop; Arbitro.EnviaMano(MiNombre, Mano); -- Comprueba su estado, si sigue jugando, si ha perdido o -- si ha ganado y deja de jugar Arbitro.SiguesJugando(MiNombre, YoSigo, HuboGanador); if YoSigo = SIGUES_JUGANDO then Terminar := False; else if YoSigo = NO_SIGUES_JUGANDO then Terminar := True; else -- Ha perdido Put_Line(“Jugador " & MiNombre'Img & ": He perdido, tengo que pagar :_("); end if; end if; if HuboGanador then NumJug := NumJug - 1; end if; end loop; end if; end Jugador; Jugadores : array (1..MAX) of Jugador; NumJug : Natural; begin if Ada.Command_Line.Argument_Count /= 1 then -- Número incorrecto de parámetros Put_Line(“Uso: ./chinos <num_jugadores>"); NumJug := 1; else NumJug := Integer'Value(Ada.Command_Line.Argument(1)); if NumJug < 2 then -- Número mínimo de jugadores Put_Line(“El número de jugadores ha de ser mayor que 1.” & NumJug'Img & " no es mayor que 1”); Put_Line(“Seleccione un valor mayor o igual que 2”); NumJug := 1; end if; if NumJug > MAX then -- Número máximo de jugadores Put_Line(NumJug'Img & " es mayor que " & MAX'Img); Put_Line(“Seleccione un valor menor o igual que " & MAX'Img); NumJug := 1; end if; end if; Arbitro.FijaNumeroJugadores(NumJug); -- Por si nos intentan colar algún valor no válido exception when Constraint_Error => NumJug := 1; Arbitro.FijaNumeroJugadores(NumJug); Put_Line(“El Valor Introducido no es correcto.”); Put_Line(“Uso: ./chinos <num_jugadores>"); end Chinos;

49.2 Manual de referencia de Ada

Capítulo 50

Programación en Ada/GLADE

50.1 Introducción a GNAT-GLADE

En primer lugar hay que aclarar que el nombre de esta librería puede confundir a los usuarios y programadores de GTK+ y GNOME. Existe una aplicación muy extendida para el diseño de interfaces gráficas que se llama 'Glade'. Un gran número de lenguajes de programación disponen de librerías para poder leer los ficheros de interfaces que genera Glade (C, C++, Ada, Python, Scheme, Ruby, Eiffel, etc). Pues bien, GNAT-GLADE no tiene nada que ver con esta (magnífica ;-) herramienta.

GLADE (*GNAT Library for Ada Distributed Environments*) es una extensión para GNAT, el compilador libre (licenciado bajo GPL) de Ada 95, que permite desarrollar aplicaciones distribuidas basándose en el anexo del manual de referencia de Ada: Annex E: Distributed Systems.

La base de las aplicaciones distribuidas de Ada 95 son las *particiones*. Básicamente una aplicación distribuida se compone de al menos un par de particiones.

Es posible utilizar GNAT-GLADE de dos formas diferentes:

- Con varias particiones sobre la misma máquina.
- Con varias particiones sobre diferentes máquinas que formen parte de una red de computadoras.

Desde luego resulta mucho más interesante la segunda de las opciones. Es más, para desarrollar aplicaciones con varias particiones sobre una misma máquina hay muchos casos en que sería más conveniente no utilizar GLADE y basarse únicamente en los mecanismos de concurrencia de Ada (las tareas): la aplicación será más eficiente.

50.2 ¿Cómo funciona GNAT-GLADE?

Cada una de las particiones de una aplicación basada en GNAT-GLADE, a la hora de la compilación se va a convertir en un ejecutable independiente. Cada uno de estos ejecutables serán los que se ejecuten por separado y se comuniquen entre ellos.

Existe una herramienta que facilita todo este proceso: gnatdist.

gnatdist lee un fichero de configuración en el que se especifica cómo queremos distribuir la aplicación y genera todos los ejecutables necesarios. De esta forma, es posible probar diferentes formas de distribuir una misma aplicación simplemente con lanzar gnatdist con un fichero de configuración distinto, sin necesidad de modificar el código de la aplicación.

50.3 Lenguaje de configuración de gnatdist

Las configuraciones de gnatdist se escriben en un lenguaje muy parecido a Ada. Es importante que todas las configuraciones se guarden en ficheros con extensión ".cfg".

Para lanzar la compilación de una aplicación distribuida con gnatdist únicamente es necesario ejecutar esta herramienta dándole como parámetro el nombre del fichero de configuración. Por ejemplo:

gnatdist Ejemplo_Configuracion1.cfg

50.3.1 ¿Cómo se escriben las configuraciones?

En cualquier punto de la configuración es posible usar comentarios, que al igual que en Ada se comienzan con los caracteres -- .

Los ficheros de configuración han de contener un bloque "configuration", cuyo nombre ha de coincidir, al igual que en el caso de los paquetes, con el nombre del fichero en el que se encuentra. Es decir:

configuration Ejemplo_Configuracion1 is -- -- Código de la configuración -- end Ejemplo_Configuracion1;

50.4 Primer ejemplo

El movimiento se aprende andando& así que, vamos a por el primer ejemplo.

En este ejemplo vamos a crear una pequeña aplicación compuesta únicamente de dos particiones:

- La primera de ellas es un servidor de sumas y restas (al más puro estilo RPC). En él se van a definir dos operaciones: suma y resta, que dados dos números enteros, van a devolver el resultado de aplicar la operación elegida sobre ambos.
- La segunda de ellas es un pequeño cliente que efectuará un par de operaciones para comprobar que efectivamente el servidor responde.

50.4.1 calculadora.ads

package Calculadora **is pragma** Remote_Call_Interface; **function** Sumar (Operando1, Operando2 : Integer) **return** Integer; **function** Restar (Operando1, Operando2 : Integer) **return** Integer; **end** Calculadora;

En esta definición del paquete llama la atención la instrucción 'pragma'. Un pragma es simplemente una directiva para el compilador. En concreto, en este ejemplo, el **pragma** Remote_Call_Interface hace que se exporte la interfaz del paquete para que otras particiones puedan realizar llamadas a sus funciones, es decir, básicamente una llamada RPC.

50.4.2 calculadora.adb

package body Calculadora **is function** Sumar (Operando1, Operando2 : Integer) **return** Integer **is begin return** Operando1 + Operando2; **end** Sumar; **function** Restar (Operando1, Operando2 : Integer) **return** Integer **is begin return** Operando1 - Operando2; **end** Restar; **end** Calculadora;

Este fichero es únicamente la implementación de las funciones del paquete calculadora.

50.4.3 cliente.adb

with Ada.Text_IO; **use** Ada.Text_IO; **with** Calculadora; **procedure** Cliente **is begin** Put_Line ("{{{1}}}" & Integer'Image (Calculadora.Sumar (321,123))); Put_Line ("{{{1}}}" & Integer'Image (Calculadora.Restar (321,123))); **end** Cliente;

Por último, el cliente. Este programa hace un par de llamadas a los funciones exportadas por el proceso de calculadora. Como se puede ver el código no tiene en cuenta si el proceso calculadora se encuentra corriendo en la misma máquina o en otra. Simplemente realiza llamadas a las funciones de la calculadora. De todo lo demás, que es mucho, ya se ha encargado gnatdist y se encarga Ada.

50.4.4 ejemplo.cfg

configuration ejemplo is pragma Starter (Ada); Particion1 : Partition := (Calculadora); Particion2 : Partition := (Cliente); procedure Cliente is in Particion2; end ejemplo;

Este es el fichero de configuración/compilación de gnatdist.

El pragma Starter describe como queremos que gnatdist compile el proyecto. Existen tres posibilidades: Ada, Shell y None. En el primero de los casos será uno de los ejecutables el que lance todos los demás. Mediante Shell es un *shell script* el que lanzará los procesos. Con None tendremos que lanzarlos a mano o hacer nuestro propio *script* de arranque.

A continuación se definen las dos particiones que se han utilizado en este ejemplo: una para la calculadora y la segunda para el cliente que le realiza peticiones.

Por último se especifica cuál es la parte principal (el *main*). Esta partición, lógicamente, ha de tener un *body*.

Cuidado con los nombres de los ficheros: han de coincidir con el nombre del paquete y además, han de estar en minúsculas. De no ser así gnatdist producirá un error.

50.4.5 Compilación y ejecución del programa

Para compilar la aplicación, como ya hemos visto, simplemente hay que ejecutar gnatdist:

gnatdist ejemplo.cfg

Si no ha habido ningún problema, se habrá producido una salida como esta:

gnatdist: checking configuration consistency ---------- ---------- ---- Configuration report ---- ---------- ---------- Configuration : Name : ejemplo Main : cliente Starter : Ada code Partition particion1 Units : - calculadora (rci) Partition particion2 Main : cliente Units : Name : ejemplo Main : cliente Starter : Ada code Partition particion1 Units : - calculadora (rci) Partition particion2 Main : cliente Units : - cliente (normal) ---------- ---------- gnatdist: building calculadora caller stubs from calculadora.ads gnatdist: building calculadora receiver stubs from calculadora.adb gnatdist: building partition particion1 gnatdist: building partition particion2 gnatdist: generating starter cliente

Nota: las siguientes copias de salidas de comandos son de un sistema GNU/Linux, pero en otro sistemas operativos existirá un equivalente.

En este momento ya tenemos construidos todos los ejecutables:

lrwxrwxrwx 1 alo alo 10 Oct 9 22:33 cliente -> particion2 -rwxr-xr-x 1 alo alo 3663802 Oct 9 22:33 particion1 -rwxr-xr-x 1 alo alo 3723193 Oct 9 22:33 particion2

Como podemos ver, al especificar el Pragma Starter (Ada) en el fichero de configuración, gnatdist ha generado un enlace simbólico al ejecutable que va a lanzar los demás.

En esta prueba vamos a ejecutar los dos programas en la misma máquina, más adelante se explicará cómo hacerlo en varias.

$ ./cliente Host for “particion1": localhost - Calculadora, ¿cuanto es 321+123? = 444 - Calculadora, ¿cuanto es 321-123? = 198

Ahora bien, ¿estamos seguros de que se ha ejecutado un programa paralelo? ¿estamos seguros de que en realidad no se ha enlazado el paquete Calculadora en los dos ejecutables?

Como podemos ver en los procesos, realmente se han ejecutado los dos programas:

[..] 7701 pts/10 S 0:00 _ bash 8753 pts/10 S 0:00 | _ ./cliente 8754 pts/10 S 0:00 | _ ./cliente 8755 pts/10 S 0:00 | _ ./cliente 8793 pts/10 S 0:00 | _ ./cliente [..] 8788 ? S 0:00 /home/alo/prog/glade/1/particion1 --detach --boot_location tcp://localhost:35802 8790 ? R 0:00 _ /home/alo/prog/glade/1/particion1 --detach --boot_location tcp://localhost:35802 8791 ? S 0:00 _ /home/alo/prog/glade/1/particion1 --detach --boot_location tcp://localhost:35802 8792 ? S 0:00 _ /home/alo/prog/glade/1/particion1 --detach --boot_location tcp://localhost:35802 8794 ? R 0:00 _ /home/alo/prog/glade/1/particion1 --detach --boot_location tcp://localhost:35802

Es más, si examinamos los símbolos de los dos ejecutables podemos ver como la calculadora tiene enlazadas las funciones de suma y resta:

08050500 g F .text 00000011 calculadora__restar 080504f0 g F .text 0000000d calculadora__sumar

y el cliente, además, tiene las funciones que usa gnatdist para el acceso:

080502c0 l F .text 00000018 calculadora__sumar___clean.0 080502e0 l F .text 0000005a calculadora__sumar___input27___read30.2 08050340 l F .text 0000018a calculadora__sumar___input27.1 080504d0 g F .text 000002bf calculadora__sumar 08050790 l F .text 00000018 calculadora__restar___clean.3 080507b0 l F .text 0000005a calculadora__restar___input67___read70.5 08050810 l F .text 0000018a calculadora__restar___input67.4 080509a0 g F .text 000002bf calculadora__restar

50.5 Instalación bajo Debian GNU/Linux

Instalación de GNAT (compilador de Ada 95) y GLADE para GNAT (extensión para soporte de programación distribuida).

Para realizar la instalación en el sistema es imprescindible estar identificado como usuario *root*:

apt-get install gnat gnat-glade

Instalación de la documentación de ambos paquetes. Estos paquetes son opcionales aunque muy recomendables.

apt-get install gnat-glade-doc gnat-doc

50.6 Enlaces externos

- *Ada: Anexo de sistemas distribuidos*. Alejandro Alonso. DIT/UPM.
- *RT-GLADE: Implementación Optimizada para Tiempo Real del Anexo de Sistemas Distribuidos de Ada 95*. Universidad de Cantabria
- Página oficial de la versión pública de GLADE
- Página de AdaCore sobre la versión soportada de GLADE
- GLADE User Guide

50.7 Manual de referencia de Ada

- Annex E: Distributed Systems

50.8 Autores

- Artículo original de Alvaro López Ortega que se puede encontrar en Programación distribuida con Ada 95 bajo GNU/Linux (I). El autor ha aceptado publicarlo bajo licencia GFDL, ver la autorización.

Capítulo 51

Programación en Ada/Ada 2005

La previsión es que cada 10 años se publique una revisión del estándar ISO del lenguaje Ada. En 1983 apareció la primera versión auspiciada por el ministerio de defensa de EE. UU. Doce años más tarde apareció la revisión conocida como Ada 95 que produjo muchas novedades, entre otras una implementación completa de la orientación a objetos.

Ya han pasado 10 años y un grupo de trabajo de la ISO preparó en el año 2005 la siguiente revisión del lenguaje. La profundidad de la revisión es más modesta que la anterior y de hecho está considerada técnicamente como una enmienda, no una revisión.

Para el diseño de los cambios se partió de las siguientes premisas:

1. Los cambios han de mantener o mejorar las ventajas actuales de Ada, especialmente en los dominios en los que goza de una mayor implantación que son los que necesitan seguridad y criticidad. Se espera que se mejoren aspectos de tiempo real y de sistemas de alta integridad.
2. Los cambios han de resolver ciertos problemas detectados en el uso de Ada 95.

El estándar fue definido por un grupo de expertos de ISO, ACM SigAda y Ada-Europe.

La publicación del estándar por parte de la ISO se produjo en enero de 2007. A pesar de ello la definición técnica terminó en 2005. Es por esto que los miembros del grupo dudaron si comenzar a llamar a la nueva versión Ada 2006, pero finalmente acordaron promover el nombre Ada 2005. Y decimos promover porque al fin y al cabo Ada 2005 no es un nombre oficial, el nombre oficial continúa siendo simplemente *Ada programming language*, ISO/IEC 8652:2005(E).

51.1 Cambios en el modelo OO

51.1.1 Interfaces al estilo Java

La principal novedad en el modelo de Orientación a Objetos es la incorporación de interfaces al estilo de Java. Es de notar que Ada 95 no admitía herencia múltiple por sus problemas asociados. En Java se resolvió el problema de una manera elegante con la distinción de la herencia de clases de la herencia de interfaces para los que sólo se definen y se heredan los métodos, pero no la implementación.

Este cambio, a parte de mejorar la herencia en Ada, permite mejorar la interoperabilidad de Ada con Java, C# y otros lenguajes con interfaces.

51.1.2 Notación Objeto.Método

Una de las principales diferencias entre Ada 95 y otros lenguajes orientados a objetos más populares es la llamada a métodos de un objeto.

En Java y C++:

Objeto.Método (Parámetros);

En Ada 95 (sin cláusula use):

Paquete.Método (Objeto, resto de parámetros);

La notación de Ada 95 tiene la única ventaja de que la parte imperativa y la parte orientada a objetos son más ortogonales entre sí. Sin embargo es un aspecto que le aleja de otros lenguajes orientados a objetos en cuanto a su sintaxis.

Para Ada 2005 se ha decidido adoptar la notación Objeto.Método que resultará más familiar para programadores que vengan de otros lenguajes.

Este cambio sólo afecta a los tagged types (*clases* en Ada 95) y a la notación de llamada, el resto continúa como en Ada 95. La llamada convencional se sigue admitiendo.

51.1.3 Explicitar cuándo se redefine un método

Se ha añadido la posibilidad de definir un método como **overriding** o **not overriding**. Esto proporciona un nivel de seguridad contra futuros cambios en una clase base como eliminar un método que ha sido redefinido por alguna clase derivada. También prevendrá el problema de querer redefinir un método y por escribirlo mal, se cree un nue-

vo método en vez de redefinirlo. Este descuido nos puede hacer perder horas hasta encontrar que no se llama el método de la clase derivada en vez de la clase base porque hemos puesto Annadir en vez de Anyadir. Además el explicitarlo servirá como documentación para comprender la derivación de clases.

51.2 Cambios en los tipos puntero

Los punteros en Ada 95 sólo podían ser de un tipo anónimo en los parámetros de subprogramas. Es decir, esta definición era ilegal:

type A **is** x; **type** A_Access **is access** A; **type** B **is record** Campo_Puntero : **access** A; -- *NO, era obligatorio A_Access* **end record**;

Ahora se admite un puntero de tipo anónimo en casi cualquier sitio donde se admite un tipo. Esto eliminará la proliferación de tipos puntero nombrados y las consiguientes conversiones de tipo entre ellos.

También es posible definir un tipo de puntero que apunta a constante y si el tipo admite el valor null o no.

type A **is** x; **type** B **is access constant** A; **type** C **is access not null** A;

Especialmente útil es especificar que un parámetro de tipo acceso no admite ser llamado con el valor null. Nos ayudará a definir claramente el contrato del subprograma y a detectar problemas lo antes posible.

function F (V : **access not null** A) **return** Integer; V := null; F (V); -- *Levantará Constraint_Error*

51.3 Dependencia mutua de tipos definidos en paquetes distintos

Se amplia el concepto de tipos incompletos para poder hacer tipos interdependientes definidos en paquetes distintos. Para ello se introduce la sentencia **limited with** que sólo nos permite hacer uso de los nombres de los tipos del otro paquete.

51.4 Visibilidad en la parte privada

Ahora es posible hacer visible un paquete únicamente en la parte privada de una especificación. Esto permitirá usar paquetes hijos privados en la parte privada de un paquete dentro de la misma jerarquía.

private package Wikilibros.Privado **is** ... **end** Wikilibros.Privado; **private with** Wikilibros.Privado; **package** Wikilibros.Ejemplos **is** ... -- *Aquí no es visible Wikilibros.Privado* **private** -- *Aquí podemos hacer uso de Wikilibros.Privado* **type** Mi_Ejemplo **is new** Wikilibros.Privado.Mi_Tipo; **end** Wikilibros.Ejemplos;

51.5 Inicializaciones por defecto

En un agregado será posible decir que cierto componente de un registro se rellena con el valor por defecto definido en su declaración.

51.6 Sintaxis especial para elevar excepciones con mensaje

Se ha simplificado la sintaxis para elevar una excepción con mensaje, y se ha hecho semejante a levantarla sin mensaje:

Ada 95

raise Problema_Interno; -- *Sin mensaje asociado* Ada.Exceptions.Raise_Exception (Problema_Interno'Identity, “No más identificadores libres”); -- *Con mensaje asociado*

Ada 2005

raise Problema_Interno; -- *Sin mensaje asociado* **raise** Problema_Interno **with** “No más identificadores libres"; -- *Con mensaje asociado*

51.7 Nuevos pragmas y atributos

Aparecen nuevos pragmas y otros que ya existían en algunas implementaciones como GNAT se estandarizan:

Unsuppress Para desactivar el efecto del pragma Suppress.

Assert Para realizar chequeos y asignarles un tratamiento común por configuración.

Preelaborable_Initialization Para indicar que la inicialización de un tipo es preelaborable.

No_Return Para indicar que un procedimiento no devuelve el control de modo normal.

Unchecked_Union Para importar uniones del lenguaje C.

Aparece el atributo Mod que da el módulo de un entero sin signo.

51.8 Ampliación de la biblioteca predefinida

Se incluye una biblioteca completa de contenedores al estilo de la STL de C++. Se ha estandarizado un paquete de manejo de directorios y otro de variables de entorno. Aunque estos paquetes los proveían todos los fabricantes de compiladores, la estandarización mejora la portabilidad de programas Ada entre distintos compiladores.

Así mismo se estandarizan operaciones con vectores y matrices, más operaciones sobre tiempos y fechas y algunos algoritmos de algebra lineal.

51.9 Nuevo tipo de caracteres

Aparte del tipo Character (8 bits) y Wide_Character (16 bits) de Ada 95 se añade un tercer tipo Wide_Wide_Character (32 bits) con soporte completo de Unicode.

Además los compiladores deben admitir código fuente codificado en Unicode 4.0, y los identificadores pueden contener letras en cualquier sistema alfabético.

51.10 Mejoras en tiempo real y concurrencia

Se ha incluido en el estándar el perfil de Ravenscar. Se han añadido paquetes predefinidos para controlar relojes de tiempo de ejecución.

Se pueden definir interfaces que han de implementar tareas u objetos protegidos. Esta posibilidad traza un puente entre las capacidades de concurrencia y de orientación a objetos del lenguaje.

51.11 Enlaces externos

Todo el siguiente material está en inglés:

51.11.1 Publicaciones y ponencias

- *Ada 2005: Putting it all together* (ponencia de SIGAda 2004)
- *GNAT: The road to Ada 2005* (ponencia de AdaSpain 2005)
- *GNAT and Ada 2005*, y esta ponencia en SIGAda 2004
- *An invitation to Ada 2005*, y esta ponencia en Ada-Europe 2004

51.11.2 *Ada 2005 Rationale*

El *Rationale* de Ada 2005 describe y justifica los cambios realizados al lenguaje. Está escrito por John Barnes y consta de los siguientes capítulos:

1. Introduction
2. Object Oriented Model
3. Access Types
4. Structure and Visibility
5. Tasking and Real-Time
6. Exceptions, Generics, Etc.
7. Predefined Library
8. Containers
9. Epilogue

51.11.3 Requisitos de Ada 2005

- *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment to ISO/IEC 8652* (10 octubre 2002), y la ponencia de este documento en SIGAda 2002

51.12 Manual de referencia de Ada

51.12.1 Ada 2005

- **Ada Reference Manual**, ISO/IEC 8652:1995(E) with COR.1:2001 and AMD.1:200y (borrador)
- **Annotated Ada Reference Manual**, ISO/IEC 8652:1995(E) with COR.1:2001 and AMD.1:200y (borrador con marcas de color para los cambios)
- Lista de borradores de la enmienda 1 a Ada (Ada 2005)

Capítulo 52

Programación en Ada/Unidades predefinidas

El estándar de Ada incluye una serie de **unidades predefinidas** útiles para muchas tareas comunes de programación y que ha de proporcionar todo compilador de Ada.

Algunas de estas unidades son:

- Paquete Standard
  - Paquete System
    - Paquete System.Address_To_Access_Conversions
    - Paquete System.Machine_Code
    - Paquete System.RPC
    - Paquete System.Storage_Elements
    - Paquete System.Storage_Pools
  - Paquete Ada
    - Paquete Ada.Command_Line
    - Paquete Ada.Exceptions
    - Paquete Ada.Numerics
      - Paquete Ada.Numerics.Complex_Elementary_Functions - G.1.2
      - Paquete Ada.Numerics.Complex_Types - G.1.1
      - Paquete Ada.Numerics.Discrete_Random - A.5.2
      - Paquete Ada.Numerics.Elementary_Functions - A.5.1
      - Paquete Ada.Numerics.Float_Random - A.5.2
      - Paquete Ada.Numerics.Generic_Complex_Elementary_Functions - G.1.2
      - Paquete Ada.Numerics.Generic_Complex_Types - G.1.1
      - Paquete Ada.Numerics.Generic_Elementary_Functions - A.5.1
    - Paquete Ada.Real_Time
    - Paquete Ada.Strings
      - Paquete Ada.Strings.Fixed
      - Paquete Ada.Strings.Bounded
      - Paquete Ada.Strings.Unbounded
    - Paquete Ada.Text_IO
      - Paquete Ada.Text_IO.Editing
    - Paquete Ada.Float_Text_IO
    - Paquete Ada.Integer_Text_IO
    - Paquete Ada.Sequential_IO
    - Función genérica Ada.Unchecked_Conversion
    - Procedimiento genérico Ada.Unchecked_Deallocation
  - Paquete Interfaces
    - Paquete Interfaces.C
      - Paquete Interfaces.C.Strings
      - Paquete Interfaces.C.Pointers
    - Paquete Interfaces.COBOL
    - Paquete Interfaces.Fortran

52.1 Manual de refencia de Ada

- Annex A: Predefined Language Environment

Capítulo 53

Programación en Ada/Recursos en la Web

Esta apéndice del wikilibro «Programación en Ada» es un listado de enlaces web ordenados por categorías.

53.1 Información general, portales

- Portal y foro del canal #Ada
- AdaPower.com
- Ada World
- Ada Resource Assoc.'s Ada Information Clearinghouse

53.2 Asociaciones y grupos

- Asociación Ada-Spain
- Asociación Ada-Europe
- Ada-Belgium Organization (en inglés)
- Ada in Denmark (en inglés)
- ACM - SIGAda: Special Interest Group on Ada

53.3 Estándar

- Ada 2005 Reference Manual
- Ada 2005 Rationale

53.3.1 Estándar de Ada 95

- Ada 95 Reference Manual
- Ada 95 Rationale

53.3.2 Estándar de Ada 83

- Ada 83 Reference Manual
- Ada 83 Rationale
- Ada '83 Quality and Style: Guidelines for Professional Programmers

53.4 Cursos y tutoriales

- Guía mínima de Ada, Guía de referencia básica de Ada 95 y Guía básica de Ada 2005 de la Universidad de las Palmas de Gran Canaria.
- Transparencias de Ada de Juan Antonio de la Puente de la UPM.
- Programación distribuida con Ada95 bajo GNU/Linux (I)
- Programación multitarea con Ada de la Universidad de Valladolid.
- Trabajo sobre Ada en el rincondelvago.com
- Trabajo en monografias.com
- Lovelace Tutorial: tutorial interactivo.

53.5 Libros electrónicos completos

- computer-books.us - Libros electrónicos de Ada 95, completos y gratuitos
- «The Big Book of Linux Ada Programming»; Copia en OOPWeb.com
- «Ada Lecture Notes»
- «Ada for Software Engineers», M. Ben-Ari

53.6 Software libre o gratuito

53.6.1 Repositorios de proyectos

- SourceForge.net: Software Map: Ada
- Libre Site: GNAT, GtkAda, XmlAda, AWS, etc
- AdaPower: Active Ada Projects You Can Join
- Google Code
- BerliOS Developer: Software Map
- Ada and Software Engineering Library 2 (ASE2)

53.6.2 Buscadores y directorios de código Ada

- Ohloh
- Koders

53.6.3 Entornos de desarrollo integrado

- GPS (Libre Site) - GNAT Programming Studio
- ada-mode para Emacs.
- AdaGIDE - Ada GNAT Integrated Development Environment for Windows
- GNAVI - The GNU Ada Visual IDE
- GRASP - Graphical Representations of Algorithms, Structures and Processes

53.6.4 Herramientas para programadores

- AdaBrowse - An Ada-To-HTML generator
- C2Ada - a translator from C to Ada - Translating C to Ada (article)
- Auto_Text_IO, generación de código de impresión de tipos Ada.
- ICC Ada ICCFMT Info page (pretty printer)
- Adalog Components, herramientas de análisis de código, paquetes de uso general y Adapplets.

53.6.5 Librerías y bindings

- XML4Ada95 - An XML package for Ada 95
- The web page of Gautier de Montmollin - Software
- Paul Pukite's Home Page, varias librerías y artículos.
- OpenALada, Ada binding for OpenAL (Open Audio Library)
- Bindings para Qt: Qt4Ada y QtAda.

53.6.6 Otras aplicaciones

- MaRTE OS Home Page, sistema operativo de tiempo real de la Universidad de Cantabria.
- Lovelace, proyecto para crear un sistema operativo completo basado en Ada.

53.7 Vendedores de compiladores

- AdaCore - GNAT
- Aonix - ObjectAda
- DDC-I
- IBM Rational Ada Developer
- Ada for .NET

53.8 Artículos

- Artículo de Ada en Wikipedia
- A Detailed Description of the GNU Ada Run Time
- Mapping UML To Ada
- Embedded.com - Ada's Slide into Oblivion
- Markus Kuhn's Ada95 Page (Why Ada?)
- Ian Sharpe - Technical Notes - Shared Libraries & Ada
- There's Still Some Life Left in Ada. When it comes to survival of the fittest, Ada ain't no dinosaur
- From extended Pascals to Ada 95
- Linux Journal - Elegance of Java and the Efficiency of C++--It's Ada
- Linux Journal - Testing Safety-Critical Software with AdaTEST
- Linux Gazette - Introduction to Programming Ada
- Linux Gazette - Gnat and Linux: C++ and Java Under Fire
- STSC CrossTalk - Ada in the 21st Century - Mar 2001
- Robotics with Ada95

53.9 Noticias y bitácoras

- Bitácora de Gneuromante - Barrapunto
- Barrapunto | GNU Visual Debugger en Ada95. (2000/12/13)
- Barrapunto de fernand0 | Información sobre Ada (2001/06/18)
- Barrapunto de fernand0 | GNADE, el entorno de bases de datos para Ada (2001/06/18)
- Barrapunto de fernand0 | GNAT comienza a integrarse con el GCC (2001/10/03)
- Barrapunto de hacking | ¿Para qué sirve Ada? (2002/04/08)
- Barrapunto de fernand0 | Comparando Ada y Java (2002/05/16)
- Barrapunto | Las tripas de un RTS libre (2002/07/05)
- Barrapunto | Liberado libro completo sobre el runtime de GNAT (2002/09/03)
- Libertonia || ¿Qué fue del Ada? (2002/09/14)
- Barrapunto | Sale el primer entorno de desarrollo libre de Ada (2003/07/22)
- Barrapunto | Ada y el software libre: ¿un futuro en común? (2004/11/2)
- Bitácora de mig21 (7781) | ¿Queda algo de vida en Ada? (2004/11/15)
- Barrapunto | Ada 2005, cada vez más cerca (2006/06/14)

53.10 Directorios de enlaces

- Ada en dmoz.org, directorio de Internet, rama en español
- Ada en dmoz.org, directorio de Internet, rama en inglés
- Ada95 y GNAT - unizar.es
- Enlaces Ada - ulpgc.es
- Resources on Ada: Managing Complexity
- Ada on Windows by Sergio Gomez

53.11 Foros, chats y listas de correo

- Lista de correo sobre Ada en español
- Canal del IRC Hispano (en español)
- Foro del canal #ada
- Foro de SóloCódigo
- Foro de “La Web del Programador”
- Usenet: comp.lang.ada
- Canal IRC de Ada en freenode.net (en inglés)

53.12 Ingeniería de software

- Ada 95 QUALITY AND STYLE Guide
- Ada 95 QUALITY AND STYLE Guide (formato PDF, PS y Word)

53.13 Buscadores

- Search Ada sites on the Internet
- Buscar en el Annotated Ada Reference Manual (AARM)
- koders.com - Buscador de software libre que permite buscar código escrito en Ada

Capítulo 54

Programación en Ada/Subprogramas

En *Ada*, los **subprogramas** se dividen en dos categorías: procedimientos y funciones. Los procedimientos son llamados como sentencias y no devuelven resultado, mientras que las funciones son llamadas como componentes de expresiones y devuelven un resultado.

54.1 Procedimientos

La llamada a un procedimiento en *Ada* constituye por sí misma una sentencia. Su especificación se puede describir de la siguiente manera:

especificación_procedimiento ::= **procedure** identificador [(especificación_parámetro { ; especificación_parámetro })] especificación_parámetro ::= identificador { , identificador } : [modo] tipo [:= expresión] modo ::= **in** | **out** | **in out**

El cuerpo del procedimiento sería:

cuerpo_procedimiento ::= especificación_procedimiento **is** [parte_declarativa] **begin** secuencia_de_sentencias [**exception** manejador_de_excepción { manejador_de_excepción }] **end** [identificador] ;

Los parámetros que se le pueden pasar a un procedimiento pueden ser de tres modos diferentes:

- **in**: el parámetro formal es una constante y permite sólo la lectura del valor del parámetro real asociado.
- **in out**: el parámetro formal es una variable y permite la lectura y modificación del valor del parámetro real asociado.
- **out**: el parámetro formal es una variable y permite únicamente la modificación del valor del parámetro real asociado.

Ninguno de estos modos implica el uso de un paso de parámetro por valor o por referencia. El compilador es libre de elegir el paso más adecuado de acuerdo al tamaño del parámetro y otras consideraciones. Como excepción, los tipos etiquetados se pasan siempre por referencia.

Por ejemplo:

```
procedure Una_Prueba (A, B: in Integer; C: out Integer)
is begin C:= A + B; end Una_Prueba;
```

Cuando se llame al procedimiento con la sentencia Una_Prueba (5 + P, 48, Q); se evalúan las expresiones 5 + P y 48 (sólo se permiten expresiones en el modo in), después se asignan a los parámetros formales A y B, que se comportan como constantes. A continuación, se asigna el valor A + B a la variable formal C. Obsérvese que especificando el modo out no se puede conocer el valor del parámetro real (Q). En este caso, el parámetro formal C es una nueva variable cuyo valor se asignará al parámetro real (Q) cuando finalice el procedimiento. Si se hubiera querido obtener el valor de Q, además de poder modificarlo, se debería haber empleado C: in out Integer.

Indicar también que dentro de un procedimiento, se puede hacer uso de la sentencia return sin argumentos que finalizaría la ejecución del procedimiento y pasaría el control a la sentencia desde la que se llamó a dicho procedimiento. Por ejemplo, para resolver una ecuación del tipo $ax^2 + bx + c = 0$:

```
with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions; procedure Ecuación_Cuadrática (A, B, C : Float; -- Por defecto es in. R1, R2 : outFloat; Válida : out Boolean) is Z: Float; begin Z := B * B - 4.0 * A * C; if Z < 0.0 or A = 0.0 then Válida := False; -- Al ser de salida, se tienen que modificar al menos una vez. R1 := 0.0; R2 := 0.0; return; else Válida := True; R1 := (-B + SQRT (Z)) / (2.0*A); R2 := (-B - SQRT (Z)) / (2.0*A); end if; end Ecuación_Cuadrática;
```

Siendo SQRT la función de Ada.Numerics.Elementary_Functions que calcula la raíz cuadrada del parámetro pasado. Si las raíces son reales, se devuelven en R1 y R2, pero si son complejas o la ecuación degenera (A = 0), finaliza la ejecución del procedimiento después de asignar a la variable Válida el valor False, para que se controle después de la llamada al procedimiento. Nótese que los parámetros out tienen que modificarse, al menos, una vez y que si no se especifica un modo, se sobreentiende que es in.

54.2 Funciones

Una función es una forma de subprograma a la que se puede invocar como parte de una expresión. Su especificación se puede describir de la siguiente manera:

especificación_función ::= **function** (identificador | símbolo_operador) [(especificación_parámetro { ; especificación parámetro })] **return** tipo especificación_parámetro ::= identificador { , identificador } : [**in**] tipo [:= expresión]

Nótese que, al contrario que los procedimientos, no se pueden pasar parámetros a la función de otro modo que no sea de entrada (in) ya que no se puede especificar otro modo (el de entrada es por defecto y obligatorio). En este sentido las funciones de Ada se parecen más a las funciones matemáticas que las funciones de otros lenguajes.

La especificación de la función es necesaria para mostrar al exterior toda la información necesaria para poder invocarla.

El cuerpo de la función sería:

cuerpo_función ::= especificación_función **is** [parte_declarativa] **begin** secuencia_de_sentencias [**exception** manejador_de_excepción { manejador_de_excepción }] **end** [identificador | símbolo_operador] ;

Un ejemplo de cuerpo de función puede ser:

function Mínimo (A, B: Integer) **return** Integer **is begin if** A > B **then return** B; **else return** A; **end if**; **end** Mínimo;

Los parámetros formales de una función se comportan como constantes locales cuyos valores son proporcionados por los parámetros reales correspondientes.

La sentencia return se utiliza para indicar el valor devuelto por la llamada a la función y para devolver el control a la expresión que llamó a la función. La expresión de la sentencia return es de complejidad arbitraria y debe ser del mismo tipo que se declara en la especificación de la función. Si se utiliza un tipo incompatible, el compilador da error. Si no se cumplen las restricciónes de un subtipo, como un rango, se eleva la excepción Constraint_Error.

El cuerpo de la función puede contener varias sentencias return y la ejecución de cualquiera de ellas terminará la función devolviendo el control a la sentencia que la había invocado. Si el flujo del programa sigue varios caminos dentro de la función hay que asegurarse de que se termine siempre con una sentencia return en cada uno de ellos. Si en tiempo de ejecución se llega al final de una función sin haber salido por un return, se levanta la excepción Program_Error. Así pues, el cuerpo de una función deberá tener al menos una sentencia return obligatoriamente.

Toda llamada a una función produce una nueva copia de cualquier objeto declarado dentro de ella, incluyendo los parámetros. Cuando la función finaliza, desaparecen sus objetos. Por tanto, es posible utilizar llamadas recursivas a una misma función, como ejemplo, se muestra una posible implementación de la función factorial:

function Factorial (N: Positive) **return** Positive **is begin if** N = 1 **then return** 1; **else return** (N * Factorial (N - 1)); **end if**; **end** Factorial;

Si se intenta evaluar la expresión Factorial (4); se llamará a la función con el parámetro 4 y dentro de la función se intentará evaluar la expresión N * Factorial (3) con lo que se volvería a llamar a la función, pero en este caso el parámetro N sería 3 (por cada llamada se realiza una copia de los parámetros) y así sucesivamente hasta que se evalúe N con valor 1 que finalizará la función y se empezarán a completarse las expresiones en sentido inverso.

Un parámetro formal de una función puede ser de cualquier tipo, incluyendo vectores o registros. Sin embargo, no puede ser un tipo anónimo, es decir, debe declararse antes, por ejemplo:

type TVector **is array** (Positive **range** <>) **of** Float; **function** Suma_Componentes (V: TVector) **return** Float **is** Resultado: Float := 0.0; **begin for** I **in** V'Range **loop** Resultado := Resultado + V(I); **end loop**; **return** Resultado; **end** Suma_Componentes;

En este ejemplo, se puede utilizar la misma función para cualquier vector de una dimensión, no importa el número de componentes del vector. Así pues, no hay límites estáticos en los parámetros pasados a las funciones. Por ejemplo, se puede utilizar de la siguiente forma:

V4: TVector(1..4) := (1.2, 3.4, 5.6, 7.8); Suma: Float; Suma := Suma_Componentes (V4);

De igual manera, una función también puede devolver un tipo del que no se conocen a priori sus límites. Por ejemplo:

function Invierte_Componentes (V: TVector) **return** TVector **is** Resultado: TVector(V'Range); -- *Fija el límite del vector a devolver.* **begin for** I **in** V'Range **loop** Resultado(I) := V(V'First + V'Last - I); **end loop**; **return** Resultado; **end** Invierte_Componentes;

La variable Resultado tiene los mismo límites que V, con lo que siempre se devuelve un vector de la misma dimensión que el pasado como parámetro.

Indicar también que una función devuelve un valor que puede utilizarse sin necesidad de realizar una asignación a una variable, con lo que se puede hacer referencia directamente, por ejemplo, a Invierte_Componentes(V4)(1), con lo que se obtendría el primer componente del vector devuelto por la función (en este caso 7.8).

54.3 Parámetros nombrados

Destacar, que tanto en procedimientos como en funciones, se puede alterar el orden de los parámetros en la lla-

mada utilizando la notación nombrada, es decir, se especifica en la llamada el nombre del parámetro real seguido del símbolo => y después el parámetro formal. Por ejemplo:

Ecuación_Cuadrática (Válida => OK, A => 1.0, B => 2.0, C => 3.0, R1 => P, R2 => Q); F := Factorial (N => (3 + I));

Esto implica conocer el nombre de los parámetros formales que, en principio, bastaría con mirar la especificación del paquete. Como se asigna uno a uno el parámetro formal al real, no hay ningún problema de ambigüedad. Conviene recordar que el parámetro formal va a la izquierda del símbolo =>, por si se da la situación de que la variable utilizada como parámetro real se llame de igual manera que el formal (no habría ambigüedad ya que están en distinto ámbito).

54.4 Parámetros por defecto

Por otra parte, se pueden establecer parámetros formales, tanto en procedimientos como en funciones, que tengan valores por defecto y, por tanto, se pueden obviar en la llamada al subprograma. Por ejemplo:

procedure Prueba_Por_Defecto (A : in Integer := 0; B: **in** Integer := 0);

Se puede llamar de estas formas:

Prueba_Por_Defecto (5, 7); -- *A = 5, B = 7* Prueba_Por_Defecto (5); -- *A = 5, B = 0* Prueba_Por_Defecto; -- *A = 0, B = 0* Prueba_Por_Defecto (B => 3); -- *A = 0, B = 3* Prueba_Por_Defecto (1, B => 2); -- *A = 1, B = 2*

En la primera sentencia, se utiliza una llamada “normal” (con notación posicional), en la segunda, se utiliza posicional y por defecto, la tercera utiliza todos los parámetros por defecto, en la cuarta nombrada y por defecto, y, por último, la quinta utiliza notación posicional y nombrada.

54.5 Manual de referencia de Ada

- Section 6: Subprograms

Capítulo 55

Programación en Ada/Unidades predefinidas/Standard

55.1 Paquete Standard

La unidad *Standard* es un paquete especial que contiene las declaraciones de todos los tipos predefinidos tales como Integer o Boolean.

En términos de compilación es como si todas las unidades tuviesen escrito antes de nada:

with Standard; **use** Standard;

También posee un paquete anidado llamado ASCII, que contiene las constantes que definen los caracteres tales como CR (retorno de carro) o LF (avance de línea). Escribiendo **use** ASCII; se pueden referenciar simplemente como CR. Sin embargo el estándar de Ada 95 no nos recomienda usar el paquete ASCII (caracteres de 7 bits), porque éste queda obsoleto tras la definición del paquete Ada.Characters.Latin_1, que define el juego completo de caracteres ISO Latín 1 de 8 bits.

55.2 Especificación

La definición del paquete Standard según el manual de referencia de Ada:

package Standard is pragma Pure(Standard); type Boolean is (False, True); -- The predefined relational operators for this type are as follows: -- function "=" (Left, Right : Boolean'Base) return Boolean; -- function "/=" (Left, Right : Boolean'Base) return Boolean; -- function "<" (Left, Right : Boolean'Base) return Boolean; -- function "<=" (Left, Right : Boolean'Base) return Boolean; -- function ">" (Left, Right : Boolean'Base) return Boolean; -- function ">=" (Left, Right : Boolean'Base) return Boolean; -- The predefined logical operators and the predefined logical -- negation operator are as follows: -- function “and” (Left, Right : Boolean'Base) return Boolean; -- function “or” (Left, Right : Boolean'Base) return Boolean; -- function “xor” (Left, Right : Boolean'Base) return Boolean; -- function “not” (Right : Boolean'Base) return Boolean; -- The integer type root_integer is predefined. -- The corresponding universal type is universal_integer. type Integer is range "implementation-defined"; subtype Natural is Integer range 0 .. Integer'Last; subtype Positive is Integer range 1 .. Integer'Last; -- The predefined operators for type Integer are as follows: -- function "=" (Left, Right : Integer'Base) return Boolean; -- function "/=" (Left, Right : Integer'Base) return Boolean; -- function "<" (Left, Right : Integer'Base) return Boolean; -- function "<=" (Left, Right : Integer'Base) return Boolean; -- function ">" (Left, Right : Integer'Base) return Boolean; -- function ">=" (Left, Right : Integer'Base) return Boolean; -- function "+" (Right : Integer'Base) return Integer'Base; -- function "-" (Right : Integer'Base) return Integer'Base; -- function “abs” (Right : Integer'Base) return Integer'Base; -- function "+" (Left, Right : Integer'Base) return Integer'Base; -- function "-" (Left, Right : Integer'Base) return Integer'Base; -- function "*" (Left, Right : Integer'Base) return Integer'Base; -- function "/" (Left, Right : Integer'Base) return Integer'Base; -- function “rem” (Left, Right : Integer'Base) return Integer'Base; -- function “mod” (Left, Right : Integer'Base) return Integer'Base; -- function "**" (Left : Integer'Base; Right : Natural) -- return Integer'Base; -- The specification of each operator for the type -- root_integer, or for any additional predefined integer -- type, is obtained by replacing Integer by the name of the type -- in the specification of the corresponding operator of the type -- Integer. The right operand of the exponentiation operator -- remains as subtype Natural. -- The floating point type root_real is predefined. -- The corresponding universal type is universal_real. type Float is digits "implementation-defined"; -- The predefined operators for this type are as follows: -- function "=" (Left, Right : Float) return Boolean; -- function "/=" (Left, Right : Float) return Boolean; -- function "<" (Left, Right : Float) return Boolean; -- function "<=" (Left, Right : Float) return Boolean; -- function ">" (Left, Right : Float) return Boolean; -- function ">=" (Left, Right : Float) return Boolean; -- function "+" (Right : Float) return Float; -- function "-" (Right : Float) return Float; -- function “abs” (Right : Float) return Float; -- function "+" (Left, Right : Float) return Float; -- function "-"

(Left, Right : Float) return Float; -- function "*" (Left, Right : Float) return Float; -- function "/" (Left, Right : Float) return Float; -- function "**" (Left : Float; Right : Integer'Base) return Float; -- The specification of each operator for the type root_real, or for -- any additional predefined floating point type, is obtained by -- replacing Float by the name of the type in the specification of the -- corresponding operator of the type Float. -- In addition, the following operators are predefined for the root -- numeric types: function "*" (Left : root_integer; Right : root_real) return root_real; function "*" (Left : root_real; Right : root_integer) return root_real; function "/" (Left : root_real; Right : root_integer) return root_real; -- The type universal_fixed is predefined. -- The only multiplying operators defined between -- fixed point types are function "*" (Left : universal_fixed; Right : universal_fixed) return universal_fixed; function "/" (Left : universal_fixed; Right : universal_fixed) return universal_fixed; -- The declaration of type Character is based on the standard ISO 8859-1 character set. -- There are no character literals corresponding to the positions for control characters. -- They are indicated in italics in this definition. See 3.5.2. type Character is (nul, soh, stx, etx, eot, enq, ack, bel, --0 (16#00#) .. 7 (16#07#) bs, ht, lf, vt, ff, cr, so, si, --8 (16#08#) .. 15 (16#0F#) dle, dc1, dc2, dc3, dc4, nak, syn, etb, --16 (16#10#) .. 23 (16#17#) can, em, sub, esc, fs, gs, rs, us, --24 (16#18#) .. 31 (16#1F#) ' ', '!', '"', '#', '$', '%', '&', ''', --32 (16#20#) .. 39 (16#27#) '(', ')', '*', '+', ',', '-', '.', '/', --40 (16#28#) .. 47 (16#2F#) '0', '1', '2', '3', '4', '5', '6', '7', --48 (16#30#) .. 55 (16#37#) '8', '9', ':', ';', '<', '=', '>', '?', --56 (16#38#) .. 63 (16#3F#) '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', --64 (16#40#) .. 71 (16#47#) 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', --72 (16#48#) .. 79 (16#4F#) 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', --80 (16#50#) .. 87 (16#57#) 'X', 'Y', 'Z', '[', '\', ']', '^', '_', --88 (16#58#) .. 95 (16#5F#) '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', --96 (16#60#) .. 103 (16#67#) 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', --104 (16#68#) .. 111 (16#6F#) 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', --112 (16#70#) .. 119 (16#77#) 'x', 'y', 'z', '{', '|', '}', '~', del, --120 (16#78#) .. 127 (16#7F#) reserved_128, reserved_129, bph, nbh, --128 (16#80#) .. 131 (16#83#) reserved_132, nel, ssa, esa, --132 (16#84#) .. 135 (16#87#) hts, htj, vts, pld, plu, ri, ss2, ss3, --136 (16#88#) .. 143 (16#8F#) dcs, pu1, pu2, sts, cch, mw, spa, epa, --144 (16#90#) .. 151 (16#97#) sos, reserved_153, sci, csi, --152 (16#98#) .. 155 (16#9B#) st, osc, pm, apc, --156 (16#9C#) .. 159 (16#9F#) ' ', '¡', '¢', '£', '¤', '¥', '¦', '§', --160 (16#A0#) .. 167 (16#A7#) '¨', '©', 'ª', '«', '¬', '', '®', '¯', --168 (16#A8#) .. 175 (16#AF#) '°', '±', '²', '³', '´', 'µ', '¶', '·', --176 (16#B0#) .. 183 (16#B7#) '¸', '¹', 'º', '»', '¼', '½', '¾', '¿', --184 (16#B8#) .. 191 (16#BF#) 'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç', --192 (16#C0#) .. 199 (16#C7#) 'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï', --200 (16#C8#) .. 207 (16#CF#) 'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×', --208 (16#D0#) .. 215 (16#D7#) 'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß', --216 (16#D8#) .. 223 (16#DF#) 'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç', --224 (16#E0#) .. 231 (16#E7#) 'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï', --232 (16#E8#) .. 239 (16#EF#) 'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷', --240 (16#F0#) .. 247 (16#F7#) 'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ', --248 (16#F8#) .. 255 (16#FF#) -- The predefined operators for the type Character are the same as for -- any enumeration type. -- The declaration of type Wide_Character is based on the standard ISO 10646 BMP character set. -- The first 256 positions have the same contents as type Character. See 3.5.2. type Wide_Character is (nul, soh ... FFFE, FFFF); package ASCII is ... end ASCII; --Obsolescent; see J.5 -- Predefined string types: type String is array(Positive range <>) of Character; pragma Pack(String); -- The predefined operators for this type are as follows: -- function "=" (Left, Right: String) return Boolean; -- function "/=" (Left, Right: String) return Boolean; -- function "<" (Left, Right: String) return Boolean; -- function "<=" (Left, Right: String) return Boolean; -- function ">" (Left, Right: String) return Boolean; -- function ">=" (Left, Right: String) return Boolean; -- function "&" (Left: String; Right: String) return String; -- function "&" (Left: Character; Right: String) return String; -- function "&" (Left: String; Right: Character) return String; -- function "&" (Left: Character; Right: Character) return String; type Wide_String is array(Positive range <>) of Wide_Character; pragma Pack(Wide_String); -- The predefined operators for this type correspond to those for String type Duration is delta "implementation-defined" range "implementation-defined"; -- The predefined operators for the type Duration are the same as for -- any fixed point type. -- The predefined exceptions: Constraint_Error: exception; Program_Error : exception; Storage_Error : exception; Tasking_Error : exception; end Standard;

55.3 Manual de referencia de Ada

- A.1 The Package Standard
- J.5 ASCII
- A.3.3 The Package Characters.Latin_1

Capítulo 56

Programación en Ada/Atributos

Los atributos son operaciones predefinidas que se pueden aplicar a tipos, objetos y otras entidades. Tienen la siguiente sintaxis:

Entidad'Atributo

o bien

Tipo'Atributo(Entidad)

56.1 Atributos aplicables a tipos

Por ejemplo estos son algunos atributos aplicables a tipos (por orden alfabético):

- **Ceiling**. Se usa con la forma *X'Ceiling(Y)*, siendo *X* cualquier tipo flotante e *Y* una variable de ese tipo. Devuelve el "techo" de *Y*, es decir el valor entero más pequeño que es mayor o igual que *Y*.

- **Digits**. Representa el número de decimales de la mantisa de un tipo flotante.

- **First**. Indica el mínimo valor que puede tomar una variable de un tipo discreto, sea entero o enumeración. Se usa con la forma T'First, siendo T un tipo discreto (entero o enumeración).

- **Image**. Se usa con la forma *X'Image(Y)*, siendo *X* cualquier tipo discreto e *Y* una variable de ese tipo. Devuelve una cadena (String) con la representación del valor de la variable *Y*.

- **Last**. Indica el máximo valor que puede tomar una variable de un tipo discreto, sea entero o enumeración.

- **Pos**. Indica la posición ocupada por un determinado valor en un tipo enumeración. Por ejemplo: *TColor'Pos(ROJO)*. La primera posición se considera 0.

- **Pred**. *TDía'Pred(VIERNES)* indica el anterior valor a *VIERNES* que toma el tipo *TDía*, si no existe, se eleva la excepción *Constraint_Error*.

- **Rounding**. Se usa con la forma *X'Rounding(Y)*, siendo *X* cualquier tipo flotante e *Y* una variable de ese tipo. Devuelve el valor de *Y* redondeado al entero más cercano. Si *Y* está exactamente entre dos valores enteros, toma el valor del mayor entero (p.e, *Float'rounding(1.5)=2.0*).

- **Size**. Indica el mínimo espacio en bits en que se pueden almacenar objetos de este tipo. Técnicamente se define como lo que ocuparía un componente de un registro de este tipo cuando el registro está empaquetado (*pragma Pack*).

- **Succ**. *TColor'Succ(ROJO)* indica el siguiente valor a *ROJO* que toma el tipo *TColor*, si no existe, se eleva la excepción *Constraint_Error*.

- **Truncation**. Se usa con la forma *X'Truncation(Y)*, siendo *X* cualquier tipo flotante e *Y* una variable de ese tipo. Devuelve el valor truncado de *Y* a un valor entero.

- **Val**. Indica el valor que ocupa una determinada posición en un tipo enumeración. Por ejemplo: *COLOR'Val(1)*.

56.2 Atributos aplicables a objetos

Por ejemplo estos son algunos atributos aplicables a objetos:

- **Size**: tamaño en bits de un objeto.
- **Valid**: si tiene una representación válida para su tipo. Útil cuando se obtienen valores desde el «mundo exterior» mediante Unchecked_Conversion u otro mecanismo.
- **First, Last**: aplicados a arrays dan el primer y el último índices del array.
- **Range**: Vector'Range indica el rango que ocupa la variable Vector, es decir, equivale a Vector'First..Vector'Last. En el caso de más de una dimensión, el valor Matriz'Range(1) indica el rango de la primera dimensión.

56.3 Ejemplos

type Tipo_enumerado **is** (Enum1, Enum2, Enum3); **for** Tipo_enumerado'Size **use** 2; -- *Para representar 3 unidades necesitamos 2 bits* **type** Tipo_entero **is range** −1 .. 5; ... **pragma** Assert (Tipo_enumerado'Last = Enum3); -- *Correcto* **pragma** Assert (Tipo_entero'First = −1); -- *Correcto* **pragma** Assert (Tipo_entero'Last = 4); -- *¡¡Incorrecto!!* **pragma** Assert (Tipo_enumerado'Pred(Enum2) = Enum1); -- *Correcto* **pragma** Assert (Tipo_enumerado'Succ(Enum2) = Enum3); -- *Correcto* **pragma** Assert (Tipo_enumerado'Image(Enum1) = “Enum1”); -- *Correcto* **type** Tipo_flotante **is digits** 10 **range** 0.0..100.0; -- *10 cifras decimales en la mantisa* Var_Flotante : Float := 1.5; Var_Flotante2 : Float := 1.9; Var_Flotante3 : Float := 1.0; Var_Flotante4 : Float := −1.8; Var_Flotante5 : Float := 1.1; ... **pragma** Assert (Float'Ceiling(var_Flotante) = 2.0); -- *Correcto* **pragma** Assert (Float'Ceiling(var_Flotante2) = 2.0); -- *Correcto* **pragma** Assert (Float'Ceiling(var_Flotante3) = 1.0); -- *Correcto* **pragma** Assert (Float'Ceiling(var_Flotante3) = 2.0); -- *¡¡Incorrecto!!* **pragma** Assert (Float'Truncation(var_Flotante) = 1.0); -- *Correcto* **pragma** Assert (Float'Truncation(var_Flotante2) = 1.0); -- *Correcto* **pragma** Assert (Float'Truncation(var_Flotante3) = 1.0); -- *Correcto* **pragma** Assert (Float'Truncation(var_Flotante4) = −1.0); -- *Correcto* **pragma** Assert (Float'Rounding(var_Flotante5) = 1.0); -- *Correcto* **pragma** Assert (Float'Rounding(var_Flotante) = 2.0); -- *Correcto* A : Character := Character'Val (32) -- *A toma el valor de espacio (valor 32 en la tabla ASCII)* B : Character := ' '; -- *B también toma el valor de espacio* Resultado : Natural; ... **if not** Resultado'Valid **then** -- *'Resultado' está fuera del rango, con valor negativo* Result := Natural'First; **end if**

56.4 Manual de referencia de Ada

- 4.1.4 Attributes
- Annex K: Language-Defined Attributes

Capítulo 57

Programación en Ada/Objetos

Los **objetos** son entidades que se crean en tiempo de ejecución y contienen un valor de un determinado tipo. En Ada los objetos se clasifican en variables y constantes.

Nótese que el concepto de objeto no implica necesariamente el uso del paradigma de la orientación a objetos. Para programar orientado a objetos en Ada se utilizan los objetos de tipo etiquetado.

57.1 Variables

Una variable se introduce en el programa mediante una declaración, que se podría denotar así:

declaración_variable ::= identificador { , identificador } : tipo [:= expresión] ;

Por ejemplo:

V: Boolean := TRUE; I, J: Integer := 1; Nombre: String := “Wikilibros"; Destino_A_Calcular: Coordenadas;

57.2 Constantes

Una constante es un objeto que se inicializa a un valor cuando se declara y posteriormente no puede cambiar.

Una constante se declara igual que una variable, pero añadiendo la palabra reservada **constant**:

declaración_constante ::= identificador { , identificador } : **constant** [tipo] [:= expresión] ;

Por ejemplo:

PI: **constant** Float := 3.14159_26536;

Un tipo especial de constante es el *número nombrado* para el cual no es necesario especificar un tipo y que es equivalente a usar el literal correspondiente.

OtroPI: **constant** := 3.14; -- *En este caso es de tipo universal_float.*

57.3 Manual de referencia de Ada

- 3.3 Objects and Named Numbers

Capítulo 58

Programación en Ada/Entrada-salida

Ada tiene cinco bibliotecas predefinidas independientes para operaciones de entrada/salida. Por tanto, la lección más importante a aprender es elegir la más adecuada en cada caso.

58.1 Direct I/O

Direct I/O se usa para acceso directo a archivos que contienen únicamente elementos del mismo tipo. Con Direct_IO el cursor del archivo se puede situar en cualquier elemento de ese tipo (el concepto conocido en inglés como *random access*). El tipo de los elementos ha de ser un subtipo definitivo (*definite subtype*), es decir, un subtipo cuyos objetos tienen un tamaño definido.

58.2 Sequential I/O

Sequential I/O se usa para el acceso secuencial a archivos que únicamente contienen elementos de un tipo especificado.

Con Sequential I/O es posible elegir entre tipos definitivos y no definitivos, pero los elementos se han de leer uno tras otro.

58.3 Storage I/O

Storage I/O nos permite almacenar *un único* elemento en un *buffer* de memoria. El elemento ha de ser de un subtipo definitivo. Storage I/O se usa en la programación concurrente para trasladar elementos de una tarea a otra.

58.4 Stream I/O

Stream I/O es el paquete de entrada/salida más potente de Ada. Stream I/O permite mezclar objetos de diferentes tipos en un archivo secuencial. Para leer/escribir de/en un *stream* (un flujo de datos) cada tipo provee un atributo 'Read y otro 'Write. Estos atributos están definidos por el compilador para cada tipo que declaremos.

Estos atributos tratan los elementos como datos sin interpretar. Son ideales tanto para entrada/salida de bajo nivel como para interoperar con otros lenguajes de programación.

Los atributos 'Input y 'Output añaden información de control adicional al archivo, tal como el atributo 'First y el 'Last de un array.

En programación orientada a objetos es posible usar los atributos 'Class'Input y 'Class'Output para almacenar y recuperar correctamente un tipo concreto de la misma clase.

Stream I/O es también el paquete de entrada/salida más flexible. Todos los atributos de E/S pueden sobrescribirse con subprogramas definidos por el usuario y es posible definir nuestros propios tipos de Stream I/O usando técnicas avanzadas de orientación a objetos.

58.5 Text I/O

Text I/O probablemente sea el tipo de entrada/salida más usada. Todos los datos del archivo se representan en formato de texto legible. Text I/O provee la posibilidad de definir el *layout* de líneas y páginas, pero el estándar es texto de forma libre.

58.6 Biblioteca predefinida

Existen varios paquetes predefinidos para la entrada/salida en Ada:

- Paquete Ada.Direct_IO
- Paquete Ada.Sequential_IO
- Paquete Ada.Storage_IO
- Paquete Ada.Streams
    - Paquete Ada.Streams.Stream_IO
- Paquete Ada.Text_IO, con sus paquetes genéricos anidados: Complex_IO, Decimal_IO, Enumera-

tion_IO, Fixed_IO, Float_IO, Integer_IO y Modular_IO.

    - Paquete Ada.Text_IO.Editing

- Paquete Ada.Float_Text_IO
- Paquete Ada.Integer_Text_IO

58.7 Manual de referencia de Ada

- 13.13 Streams
- A.8.1 The Generic Package Sequential_IO
- A.8.4 The Generic Package Direct_IO
- A.10.1 The Package Text_IO
- A.12.1 The Package Streams.Stream_IO

Capítulo 59

Programación en Ada/Unidades predefinidas/Ada.Text IO

Ada.Text_IO es un paquete predefinido para la entrada y salida de texto.

Tiene procedimientos Get y Put para strings y caracteres; y varios paquetes genéricos anidados para la entrada/salida de otros tipos en formato texto: Decimal_IO, Enumeration_IO, Fixed_IO, Float_IO, Integer_IO y Modular_IO.

59.1 Ejemplo de E/S por consola

with Ada.Text_IO; **with** Ada.Characters.Handling; -- *Lee de entrada un número entero no dejando al usuario escribir -- caracteres que no sean dígitos. Esta versión sólo funciona -- correctamente en Windows. Para que funcione en Linux y otros -- sistemas Unix, hay que cambiar los valores de Intro y Back.* **procedure** Leer_Entero **is** -- *Para Linux cambiar por ASCII.LF* Intro : **constant** Character := ASCII.CR; -- *Para Linux cambiar por ASCII.Del* Back : **constant** Character := ASCII.BS; Char : Character; Fin : Boolean := False; Número : Natural := 0; -- *Cadena para leer el número carácter a carácter -- El máximo de caracteres es Integer'Width - 1 porque no leemos signo* Cadena_Número : String (1 .. Integer'Width - 1); **begin** Ada.Text_IO.Put ("Escriba un número y pulse Enter: "); **while not** Fin **loop** Ada.Text_IO.Get_Immediate (Char); **if** Ada.Characters.Handling.Is_Digit (Char) **then** Número := Número + 1; Cadena_número(Número) := Char; Ada.Text_IO.Put (Char); **elsif** Char = Intro **then** Fin := True; **elsif** Número>0 **and** Char = Back **then** -- *Si el usuario ha pulsado la tecla backspace -- borra el dígito escrito anteriormente* Ada.Text_IO.Put (ASCII.BS & ' ' & ASCII.BS); Número:=Número-1; **end if**; **end loop**; Número := Integer'Value (Cadena_Número (1 .. Número)); Ada.Text_IO.New_line; Ada.Text_IO.Put_Line ("Has escrito:" & Integer'Image (Número)); **exception when** Constraint_Error => Ada.Text_IO.New_line; Ada.Text_IO.Put_Line ("Lo siento: " & Cadena_Número & " es demasiado largo para almacenarse"); **end** Leer_Entero;

59.2 Ficheros de texto

Ada.Text_IO también permite el acceso y modificación de ficheros de texto de forma secuencial.

La mayoría de funciones de Ada.Text_IO están disponibles para ser usadas con ficheros de texto. Para eso, se usan variables de tipo File_type, necesarias para especificar a qué archivo acceder. Muchas de las funciones conocidas para consola de **Ada.Text_IO** se pueden usar en archivos de texto pasando por parámetro una variable de tipo File_type.

Algunas funciones y procedimientos para el manejo de ficheros con **Ada.Text_IO**:

Open(F,Modo,Ruta) Permite abrir un fichero. Si el fichero no existe, devuelve una excepción 'Name_error'. 'F' es una variable File_type, 'Ruta' es la ruta del sistema donde se localiza el fichero y 'Modo' especifica como abrir el fichero: 'In_file' significa lectura, 'Out_file' significa escritura (borrando lo anterior) y 'Append_file' significa escritura empezando desde el final. Para acceder a este archivo, lo haremos a través de la variable File_type 'F'.

Create(F,Modo,Ruta) Crea un fichero en la ruta del sistema elegida. Si no existe, se crea, y si existe, se sobreescribe. Los parámetros son los mismos que en 'Open', pero por defecto el modo es 'Out_file' (si creas un archivo, suele ser para escribir en él). Para acceder a este archivo, lo haremos a través de la variable File_type 'F'.

Close(F) Cierra el archivo referido por la variable 'F'. Es necesario hacer esto cuando dejemos de leer o escribir en un fichero.

Get(F,C) Lee un carácter de un fichero, siendo F una variable File_type y 'C' una variable character. Para leer se debe haber hecho un 'Open' previamente.

Put(F,C) Escribe un carácter en un fichero, siendo F una variable File_type y 'C' una variable Character. Pa-

ra escribir se debe haber hecho un 'Open' en modo escritura o un 'Create' previamente.

End_of_file(F) Esta función devuelve 'True' (*boolean*) si hemos llegado al final del fichero y 'False' si quedan elementos por leer. Es importante saber usar esta función, ya que si intentamos leer un elemento del fichero habiendo llegado al final de éste, saltará la excepción 'End_Error'.

End_of_line(F) Esta función devuelve 'True' (boolean) si hemos llegado al final de la linea actual y 'False' si quedan elementos por leer.

Reset(F) Reinicia el cursor; para saber que elemento estamos leyendo, se guarda un cursor con la posición del elemento actual; esta operación lo reinicia, como si cerrásemos el fichero y lo volviésemos a abrir.

Para encontrar la lista completa de operaciones sobre ficheros de **Ada.Text_IO**, se puede ver en el manual de referencia: A.10.1 Text_IO.

59.3 Ejemplo de E/S por fichero

with Ada.Command_Line,Ada.Text_Io; **use** Ada.Command_Line,Ada.Text_Io; **procedure** Visor_texto **is** -- *Lee de un fichero de texto cuya ruta se pasa por parámetro o se pregunta -- explícitamente, y se visualiza por pantalla, tomando como 'estandard' una consola -- de 24 líneas de largo y 80 caracteres de ancho* Caracteres_Por_Linea : **constant** Natural := 79; Lineas_Por_Pantalla : **constant** Natural := 24; F : File_Type; Linea : String (1 .. Caracteres_Por_Linea); Indice : Natural; Contador : Natural := 0; **procedure** Esperar_Tecla **is** C : Character; **begin** Get_Immediate(C); **end** Esperar_Tecla; **begin if** Argument_Count>0 **then** -- *Si hay parametros, usamos el primero como ruta del archivo* Open(F,In_File,Argument(1)); **else** -- *Si no hay parámetros, preguntamos explícitamente la ruta del archivo* Put_Line("Introduzca la ruta del archivo a abrir: "); Get_Line(Linea,Indice); Open(F,In_File,Linea(1..Indice)); New_Line(3); **end if**; Put_Line("------------------------------------"); Put_Line("- Visor de texto - " & Name(F)); Put_Line("------------------------------------"); -- *La función Name() nos devuelve la ruta del archivo* New_Line(2); **while not** End_Of_File(F) **loop** -- *Leemos hasta llegar al final del fichero -- Si llegamos al final e intentamos leer, dará error, por lo que hay que prevenirlo* **if** Contador>=Lineas_Por_Pantalla-2 **then** New_Line; Put_Line("---- Presione una tecla para continuar"); Esperar_Tecla; New_Line; Contador:=0; **end if** ; -- *Leemos una linea desde el archivo, tomando su longitud en 'Indice' -- y guardando la linea en un string* Get_Line(F,Linea,Indice); -- *Visualizamos la linea obtenida por pantalla, pero solo hasta la longitud obtenida* Put_Line(Linea(1..Indice)); Contador:=Contador+1; **end loop** ; Close(F); -- *Controlamos posibles errores que puede haber con ficheros* **exception when** Name_Error=> New_line(2); Put_Line("**** Error ****"); Put_Line("Nombre de archivo no valido"); **when** Use_Error=> New_line(2); Put_Line("**** Error ****"); Put_Line("Archivo ya abierto o inaccesible"); **end** Visor_texto;

59.4 Portabilidad

Un programa hecho en Ada usando la librería Ada.Text_IO debería poderse compilar sin problemas (si no se usan librerías propias del sistema) tanto en sistemas Windows como en sistemas Unix (incluyendo GNU/Linux) . Sin embargo, puede que su funcionamiento no sea igual en ambos sistemas si no se tienen en cuenta ciertas diferencias en sus consolas.

Por ejemplo, todos los sistemas Unix soportan el estándar de terminal ANSI, mientras que en Windows ME, NT, 2000 y XP la consola no es compatible con ANSI (aunque es posible configurarlo cargando el *driver* ANSI.SYS). Además el salto de linea es diferente en Unix y en Windows: en sistemas Unix consta de un carácter y en Windows de dos. A la hora de programar, has de tener en cuentas algunas diferencias:

- El salto de línea en sistemas Unix es ASCII.LF. En Windows es ASCII.LF & ASCII.CR. Sin embargo usando Get_Line, End_Of_Line y Put_Line se asegura portabilidad en este aspecto.

- Algunas funciones de **Ada.Text_IO** no funcionan correctamente en Windows, como el procedimiento *New_Page*. Get_immediate(C,B) también puede dar problemas con algunas teclas en Linux.

- El carácter de control asociado a ciertas teclas puede variar entre Unix y Windows. Por tanto si tu programa intenta detectar pulsaciones de teclas específicas, como 'Enter', etc. deberás adecuar el carácter con el que se identifica la tecla. Algunos ejemplos:

Intro: character:=ASCII.Cr; -- *Windows* Intro: character:=ASCII.Lf; -- *Unix* Backspace: character:=ASCII.Bs; -- *Windows (tecla de borrar)* Backspace: character:=ASCII.Del; -- *Unix*

Debes tener esto en cuenta si deseas hacer portable tu programa, y que funcione tanto en Unix como en Windows. Por lo tanto recomendamos que pruebes la aplicación en ambos sistemas si es posible, para asegurar su buen funcionamiento. Si deseas más funciones para la consola, y ver sus diferentes implementaciones en Linux y Windows, puedes mirar el paquete Pantalla_ansi.

59.5 Manual de referencia de Ada

- A.10.1 The Package Text_IO

Capítulo 60

Programación en Ada/Unidades predefinidas/Ada.Float Text IO

Ada.Float_Text_IO es un paquete predefinido de la biblioteca estándar para la entrada/salida de datos de tipo Float.

Lo primero que hay que hacer es importar este paquete:

with Ada.Float_Text_IO; **use** Ada.Float_Text_IO;

Están predefinidos dentro de esta biblioteca los siguientes procedimientos:

Put (Item, Fore, Aft, Exp); Put (File, Item, Fore, Aft, Exp);

Donde:

- *Item* es el número en coma flotante a mostrar.
- *Fore* el número de caracteres antes del punto. Si es mayor que el número de dígitos, se rellenan con espacios. Este parámetro es ideal para justificar varios números en una columna.
- *Aft* cuantos dígitos decimales queremos que se muestren.
- *Exp* cuantos dígitos dejamos para exponente en notación científica. Si es cero no se usa la notación científica.
- *File* es un fichero abierto con Ada.Text_IO.Open para salida.

Podemos dejar parámetros en blanco y se tomarían por defecto los valores que tenga el procedimiento en la biblioteca:

Put (Número); Put (Número,X); Put (Número, ,Y,Z); -- *¡¡NO!! Error -- La siguiente es la manera correcta de dejar el número de dígitos -- decimales al valor por defecto.* Put (Item => Número, Aft => Y, Exp => Z);

Lo más usual es que usemos Put dentro de otro procedimiento, seleccionando los parámetros a nuestro gusto:

with Ada.Float_Text_IO; **use** Ada.Float_Text_IO; **procedure** Escribir_Real (X: float) **is begin** Put (X, 8, 2, 2); **end** Escribir_Real;

Si llamamos a Escribir_Real (-100.0) obtendremos por pantalla lo siguiente:

-1.00E+2

(que al ser notación científica es -1.00×10^2).

Obsérvese que el signo usa uno de los espacios en blanco con lo cual no dejamos 8 espacios entre números sino 7, usamos un blanco para el signo.

Para una representación tradicional de un número real podemos dejar el parámetro de Exp a 0 y el de Fore a 0, en tal caso no se usará notación científica ni espacios en blanco antes del número.

with Ada.Float_Text_IO; **use** Ada.Float_Text_IO; **procedure** Escribir_Decimales (X: float) **is begin** Put (X, 0, 4, 0); -- Notación tradicional con 4 decimales. **end** Escribir_Decimales;

Si llamamos a Escribir_Decimales (-125.720498) obtendremos por pantalla lo siguiente:

-125.7205

(nótese el redondeo en el último dígito).

60.1 Especificación

Este paquete es equivalente a una instanciación del paquete genérico Float_IO, anidado dentro de Ada.Text_IO.

Según el manual de referencia de Ada, la especificación de este paquete puede ser la siguiente (en rigor no se requiere que sea realmente una instanciación):

with Ada.Text_IO; **package** Ada.Float_Text_IO **is new** Ada.Text_IO.Float_IO(Float);

60.2 Manual de referencia de Ada

- A.10.9 Input-Output for Real Types

Capítulo 61

Programación en Ada/Unidades predefinidas/Ada.Text IO.Editing

Ada.Text_IO.Editing es un paquete predefinido para la entrada y salida de tipos de coma fija en formato monetario.

Ejemplo:

```
with Ada.Text_IO.Editing; procedure Ejemplo_Euros is type T_Precio_En_Euros is delta 0.01 digits 6; package Euros_IO is new Ada.Text_IO.Editing.Decimal_Output (Num => T_Precio_En_Euros, Default_Currency => "EUR ", Default_Fill => ' ', Default_Separator => '.', Default_Radix_Mark => ','); Un_Precio : constant T_Precio_En_Euros := 5873.26; begin Ada.Text_IO.Put_Line (Euros_IO.Image (Item => Un_Precio, Pic => Ada.Text_IO.Editing.To_Picture ("#_###_###_##9.99"))); end Ejemplo_Euros;
```

La salida es:

```
EUR 5.873,26
```

61.1 Manual de referencia de Ada

- F.3.3 The Package Text_IO.Editing

Capítulo 62

Programación en Ada/Unidades predefinidas/Ada.Sequential IO

Ada.Sequential_IO es un paquete predefinido de la biblioteca estándar para el manejo de entrada y salida de ficheros secuenciales, esto es, ficheros en los que se leen o escriben los datos uno detrás de otro.

62.1 Instanciación

Para ser usado, el paquete **Ada.Sequential_IO** debe ser instanciado con el tipo a usar, es decir se debe especificar qué elemento debe leer/escribir: un entero, un carácter, un booleano, o un tipo definido por nosotros.

Para ello, usaremos la cláusula with. A continuación procedemos a instanciar el paquete de la siguiente forma

package Nombre **is new** Ada.Sequential_IO(Tipo);

Ahora podremos usar la cláusula use.

use Nombre

O bien ponerlo como prefijo a todas las operaciones y tipos de **Ada.Sequential_IO**

Nombre.Open(F,Nombre.In_file,"archivo.dat");

Si se hacen varias instanciaciones de **Ada.Sequential_IO** a diferentes tipos, será necesario usar el nombre del paquete que hemos puesto como prefijo

package Pkg_integer **is new** Ada.Sequential_IO(Integer) **package** Pkg_character **is new** Ada.Sequential_IO(Character);; F: Pkg_integer.File_type; G: Pkg_character.File_type;

62.2 Funciones y procedimientos más comunes

Open(F,Modo,Ruta) Permite abrir un fichero. Si el fichero no existe, devuelve una excepción 'Name_error'. 'F' es una variable File_type, 'Ruta' es la ruta del sistema donde se localiza el fichero y 'Modo' especifica como abrir el fichero: 'In_file' significa lectura, 'Out_file' significa escritura (borrando lo anterior) y 'Append_file' significa escritura empezando desde el final. Para acceder a este archivo, lo haremos a través de la variable File_type 'F'.

Create(F,Modo,Ruta) Crea un fichero en la ruta del sistema elegida. Si no existe, se crea, y si existe, se sobreescribe. Los parámetros son los mismos que en 'Open', pero por defecto el modo es 'Out_file' (si creas un archivo, suele ser para escribir en él). Para acceder a este archivo, lo haremos a través de la variable File_type 'F'.

Close(F) Cierra el archivo referido por la variable 'F'. Es necesario hacer esto cuando dejemos de leer o escribir en un fichero.

Read(F,X) Lee de un fichero una variable 'X' del tipo instanciado, siendo F una variable File_type . Para leer se debe haber hecho un 'Open' previamente.

Write(F,X) Escribe en un fichero una variable 'X' del tipo instanciado, siendo siendo F una variable File_type. Para escribir se debe haber hecho un 'Open' en modo escritura o un 'Create' previamente.

End_of_file(F) Esta función devuelve 'True' (*boolean*) si hemos llegado al final del fichero y 'False' si quedan elementos por leer. Es importante saber usar esta función, ya que si intentamos leer un elemento del fichero habiendo llegado al final de éste, saltará la excepción 'End_Error'.

Reset(F) Reinicia el cursor; para saber que elemento estamos leyendo, se guarda un cursor con la posición del elemento actual; esta operación lo reinicia, como si cerrásemos el fichero y lo volviésemos a abrir.

La lista completa de operaciones de **Ada.Sequential_IO**, se encuentra en el manual de referencia: A.8.1 The Package Sequential_IO.

62.3 Excepciones más frecuentes

Estas son algunas de las excepciones que pueden aparecer:

1. **End_Error**: Hemos intentado leer un fichero cuando éste estaba vacío o si hemos llegado al final de él. Para prevenirlo, hay que usar la función *End_of_file*
2. **Name_Error**: La ruta con la que hemos abierto un fichero es incorrecta, y el archivo no existe.
3. **Use_Error**: Se suele dar cuando el archivo al que intentamos acceder ya está abierto por otro programa, o hemos hecho dos *Open* de un mismo archivo sin cerrar el primero. Para el primer caso, poco podemos hacer (es un programa externo el que influye) excepto avisar al usuario de que lo cierre; en el segundo caso el error es de código, ya que no hemos cerrado el fichero con un *Close*.

62.4 Ejemplos

```
with Ada.Text_IO, Ada.Sequential_IO; -- Ejemplo sencillo de uso Ada.Sequential_IO; -- Copiamos un archivo para hacer una copia de seguridad
procedure Copiador is type Byte is mod 2**8;
package Pkg_Byte is new Ada.Sequential_Io(Byte); C : Character := Ascii.Nul; B : Byte; Archivo_Original, Archivo_Copia : Pkg_Byte.File_Type; Ruta_Entrada : String (1 .. 32); Long_Ent : Integer := 0; begin
Ada.Text_Io.Put_Line("Introduzca el nombre del fichero de a copiar (maximo 32 caracteres)");
Ada.Text_Io.Get_Line(Ruta_Entrada,Long_Ent);
-- Abrimos el fichero en modo In_file, modo lectura, -- con la ruta especificada por el usuario
Pkg_Byte.Open(Archivo_Original,Pkg_Byte.In_File,Ruta_Entrada(1..Long_Ent));
-- Creamos un fichero del mismo nombre con terminación '.backup'
Pkg_Byte.Create(Archivo_Copia,Pkg_Byte.Out_File,Ruta_Entrada(1..Long_Ent) & ".backup"); -- Copiamos el contenido del fichero original al recién creado -- Nótese el uso de End_of_file para prevenir la lectura de final de fichero
while not Pkg_Byte.End_Of_File(Archivo_Original) loop
Pkg_Byte.Read(Archivo_Original,B);
Pkg_Byte.Write(Archivo_Copia,B); end loop ; -- Importante: ¡No hay que olvidarse de cerrar los ficheros cuando no los usemos!
Pkg_Byte.Close(Archivo_Original);
Pkg_Byte.Close(Archivo_Copia);
Ada.Text_Io.Put_Line("Operacion realizada con exito"); Ada.Text_Io.Put_Line("Presione cualquier tecla para finalizar"); Ada.Text_Io.Get_Immediate(C);
exception when Pkg_Byte.Name_Error=>
Ada.Text_Io.Put_Line("Error: Nombre de archivo o ruta incorrectos"); when Pkg_Byte.Use_Error=>
Ada.Text_Io.Put_Line("Error: El archivo ya esta abierto"); end Copiador;
```

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Sequential_IO; -- Ejemplo avanzado de uso Ada.Sequential_IO; -- Programa de 'cifrado' César -- Leemos de un fichero, hacemos un desplazamiento mediante -- una contraseña (método César) y escribimos en otro fichero. -- Evidentemente la seguridad es casi nula, basta ir probando las -- 255 posibilidades para sacarlo. Y si cifras dos veces consecutivas -- de forma que sumen 256 también se saca.
procedure Cesar is type Byte is mod 2**8;
package Pkg_Byte is new Ada.Sequential_Io(Byte); C : Character := Ascii.Nul; B : Byte; Archivo_Entrada, Archivo_Salida : Pkg_Byte.File_Type; Ruta_Entrada, Ruta_Salida : String (1 .. 32); Password, Long_Ent, Long_Sal, Aux : Integer := 0; begin while C/='c' and C/='d' loop Ada.Text_Io.Put_Line("Cifrar (c) o Descifrar (d)?"); Ada.Text_Io.Get_Immediate (C); if C/='c' and C/='d' then Ada.Text_Io.New_Line;
Ada.Text_Io.Put_Line("Error:Pulse la C o la D"); end if; end loop; Ada.Text_Io.Put("Ha elegido: "); if C='c' then
Ada.Text_Io.Put("Cifrar"); Ada.Text_Io.New_Line(2); else Ada.Text_Io.Put("Descifrar");
Ada.Text_Io.New_Line(2); end if;
Ada.Text_Io.Put_Line("Introduzca el nombre del fichero de entrada (maximo 32 caracteres)");
Ada.Text_Io.Get_Line(Ruta_Entrada,Long_Ent);
Pkg_Byte.Open(Archivo_Entrada,Pkg_Byte.In_File,Ruta_Entrada(1..Long
Ada.Text_Io.Put_Line("Introduzca el nombre del fichero de salida (maximo 32 caracteres)"); Ada.Text_Io.Put_Line("Ojo, sobreescribira el fichero sin preguntar!");
Ada.Text_Io.Get_Line(Ruta_Salida,Long_Sal);
Pkg_Byte.Create(Archivo_Salida,Pkg_Byte.Out_File,Ruta_Salida(1..Long
while Password<1 {{Ada/Reservadalor}} Password>255 loop Ada.Text_Io.Put_Line("Elija un password (numero del 1 al 255)");
Ada.Integer_Text_Io.Get(Password); if Password<1 {{Ada/Reservadalor}} Password>255 then
Ada.Text_Io.New_Line; Ada.Text_Io.Put_Line("Error: El valor no es correcto. Debe estar entre 1 y 255"); end if; end loop; while not Pkg_Byte.End_Of_File(Archivo_Entrada) loop
Pkg_Byte.Read(Archivo_Entrada,B); if C='c' then Aux:=(Integer(B)+Password) mod 256; else Aux:=(Integer(B)-Password) mod 256; end if; Pkg_Byte.Write(Archivo_Salida,Byte(Aux)); end loop; Pkg_Byte.Close(Archivo_Entrada);
Pkg_Byte.Close(Archivo_Salida);
Ada.Text_Io.Put_Line("Operacion realizada con exito"); Ada.Text_Io.Put_Line("Presione cualquier tecla para finalizar"); Ada.Text_Io.Get_Immediate(C);
exception when Pkg_Byte.Name_Error=>
Ada.Text_Io.Put_Line("Error: Nombre de archivo o ruta incorrectos"); when Pkg_Byte.Use_Error=>
Ada.Text_Io.Put_Line("Error: El archivo ya esta abierto"); when Ada.Integer_Text_IO.Data_Error=>
Ada.Text_Io.Put_Line("Error: La contraseña debe estar entre 1 y 255"); end Cesar;
```

62.5 Manual de referencia de Ada

- A.8.1 The Package Sequential_IO

Capítulo 63

Programación en Ada/Unidades predefinidas/Ada.Strings.Fixed

Ada.Strings.Fixed proporciona subprogramas para la transformación y el manejo de strings de tamaño fijo.

63.1 Ejemplo

package Fechas **is type** Fecha **is record** Dia : Positive **range** 1 .. 31; Mes : Positive **range** 1 .. 12; Año : Positive **range** 1 .. 3000; **end record**; **subtype** String_Fecha **is** String (1..10); -- *Pasa la fecha a string en formato "dd-mm-aaaa".* -- **function** Imagen (F: Fecha) **return** String_Fecha; **end** Fechas; **with** Ada.Strings.Fixed; **use** Ada.Strings.Fixed; **use** Ada.Strings; **package body** Fechas **is function** Imagen (F: Fecha) **return** String_fecha **is procedure** Mover_Imagen_Positive (N: Positive; S: **in out** String) **is begin** -- *Move copia un string en otro de otro tamaño, añadiendo un -- padding opcionalmente. Trim elimina los blancos a izquierda o -- derecha (en este caso el blanco que pone el 'Image).* Move (Source => Trim (Positive'Image (N), Left), Target => S, Justify => Right, Pad => '0'); **end** Mover_Imagen_Positive; S_Fecha : String_Fecha; **begin** Mover_Imagen_Positive (F.Dia, S_Fecha (1..2)); S_Fecha (3) := '-'; Mover_Imagen_Positive (F.Mes, S_Fecha (4..5)); S_Fecha (6) := '-'; Mover_Imagen_Positive (F.Año, S_Fecha (7..10)); **return** S_Fecha; **end** Imagen; **end** Fechas;

63.2 Especificación

Según el manual de referencia de Ada, la especificación de este paquete ha de ser la siguiente:

with Ada.Strings.Maps; **package** Ada.Strings.Fixed **is pragma** Preelaborate(Fixed); -- *"Copy" procedure for strings of possibly different lengths* **procedure** Move (Source : **in** String; Target : **out** String; Drop : **in** Truncation := Error; Justify : **in** Alignment := Left; Pad : **in** Character := Space); -- *Search subprograms* **function** Index (Source : **in** String; Pattern : **in** String; Going : **in** Direction := Forward; Mapping : **in** Maps.Character_Mapping := Maps.Identity) **return** Natural; **function** Index (Source : **in** String; Pattern : **in** String; Going : **in** Direction := Forward; Mapping : **in** Maps.Character_Mapping_Function) **return** Natural; **function** Index (Source : **in** String; Set : **in** Maps.Character_Set; Test : **in** Membership := Inside; Going : **in** Direction := Forward) **return** Natural; **function** Index_Non_Blank (Source : **in** String; Going : **in** Direction := Forward) **return** Natural; **function** Count (Source : **in** String; Pattern : **in** String; Mapping : **in** Maps.Character_Mapping := Maps.Identity) **return** Natural; **function** Count (Source : **in** String; Pattern : **in** String; Mapping : **in** Maps.Character_Mapping_Function) **return** Natural; **function** Count (Source : **in** String; Set : **in** Maps.Character_Set) **return** Natural; **procedure** Find_Token (Source : **in** String; Set : **in** Maps.Character_Set; Test : **in** Membership; First : **out** Positive; Last : **out** Natural); -- *String translation subprograms* **function** Translate (Source : **in** String; Mapping : **in** Maps.Character_Mapping) **return** String; **procedure** Translate (Source : **in out** String; Mapping : **in** Maps.Character_Mapping); **function** Translate (Source : **in** String; Mapping : **in** Maps.Character_Mapping_Function) **return** String; **procedure** Translate (Source : **in out** String; Mapping : **in** Maps.Character_Mapping_Function); -- *String transformation subprograms* **function** Replace_Slice (Source : **in** String; Low : **in** Positive; High : **in** Natural; By : **in** String) **return** String; **procedure** Replace_Slice (Source : **in out** String; Low : **in** Positive; High : **in** Natural; By : **in** String; Drop : **in** Truncation := Error; Justify : **in** Alignment := Left; Pad : **in** Character := Space); **function** Insert (Source : **in** String; Before : **in** Positive; New_Item : **in** String) **return** String; **procedure** Insert (Source : **in out** String; Before : **in** Positive; New_Item : **in** String; Drop : **in** Truncation := Error); **function** Overwrite (Source : **in** String; Position : **in** Positive; New_Item : **in** String) **return** String; **procedure** Overwrite (Source : **in out** String; Position : **in** Positive; New_Item : **in** String; Drop : **in** Truncation := Right); **function** Delete (Source : **in** String; From : **in** Positive; Through : **in** Natural) **return**

```
String; procedure Delete (Source : in out String; From : in Positive; Through : in Natural; Justify : in Alignment := Left; Pad : in Character := Space); -- String selector subprograms function Trim (Source : in String; Side : in Trim_End) return String; procedure Trim (Source : in out String; Side : in Trim_End; Justify : in Alignment := Left; Pad : in Character := Space); function Trim (Source : in String; Left : in Maps.Character_Set; Right : in Maps.Character_Set) return String; procedure Trim (Source : in out String; Left : in Maps.Character_Set; Right : in Maps.Character_Set; Justify : in Alignment := Strings.Left; Pad : in Character := Space); function Head (Source : in String; Count : in Natural; Pad : in Character := Space) return String; procedure Head (Source : in out String; Count : in Natural; Justify : in Alignment := Left; Pad : in Character := Space); function Tail (Source : in String; Count : in Natural; Pad : in Character := Space) return String; procedure Tail (Source : in out String; Count : in Natural; Justify : in Alignment := Left; Pad : in Character := Space); -- String constructor functions function "*" (Left : in Natural; Right : in Character) return String; function "*" (Left : in Natural; Right : in String) return String; end Ada.Strings.Fixed;
```

63.3 Manual de refencia de Ada

- A.4.3 Fixed-Length String Handling

Capítulo 64

Programación en Ada/Unidades predefinidas/Ada.Strings.Unbounded

Los Unbounded_String a diferencia de los strings nativos no son arrays, eso quiere decir que no se les puede aplicar el atributo 'Range ni los parentesis para indexar. En vez de eso son un tipo privado que tiene en su paquete (el Ada.Strings.Unbounded) todas las operaciones necesarias para reemplazar caracteres (Replace_Element), convertir a string (To_String) y desde string (To_Unbounded_String), concatenar (operador "&"), reemplazar subcadenas del Unbounded_String por otras (Replace_Slice), buscar subcadenas (Index), etc.

La ventaja de un Unbounded_String es que puede crecer y decrecer sin límite (o sin otro límite que la memoria disponible).

64.1 Manual de referencia de Ada

- A.4.5 Unbounded-Length String Handling

Capítulo 65

Programación en Ada/Unidades predefinidas/Ada.Command Line

Ada.Command_Line es un paquete predefinido para la consulta de la línea de comandos para los sistemas operativos que lo soportan, que son la mayoría. También permite establecer el código de error de terminación del programa.

65.1 Ejemplo

with Ada.Text_IO; **with** Ada.Command_Line; **use** Ada.Command_Line; -- *Imprime los argumentos pasados por línea de comandos* **procedure** Imprimir_Argumentos **is begin** Ada.Text_IO.Put_Line ("Imprimiendo argumentos pasados a " & Command_Name & '.'); **for** I **in** 1 .. Argument_Count **loop** Ada.Text_IO.Put_Line ("Argumento nº" & Integer'Image (I) & ": " & Argument (I)); **end loop**; Set_Exit_Status (Success); **exception when others** => Set_Exit_Status (Failure); **end** Imprimir_Argumentos;

65.2 Especificación

Según el manual de referencia de Ada (el estándar), la especificación de este paquete ha de ser la siguiente:

package Ada.Command_Line **is pragma** Preelaborate(Command_Line); **function** Argument_Count **return** Natural; **function** Argument (Number : **in** Positive) **return** String; **function** Command_Name **return** String; **type** Exit_Status **is** *implementation-defined integer type*; Success : **constant** Exit_Status; Failure : **constant** Exit_Status; **procedure** Set_Exit_Status (Code : **in** Exit_Status); **private** ... -- *not specified by the language* **end** Ada.Command_Line;

65.3 Manual de referencia de Ada

- A.15 The Package Command_Line

Capítulo 66

Programación en Ada/Operadores

66.1 Clasificación

Ésta es la lista de operadores de Ada de menor a mayor precedencia.

66.2 Propiedades

En todos los casos, excepto para la exponenciación, los dos operandos deben ser del mismo tipo.

Los operadores se pueden sobrecargar.

66.3 Comprobación de pertenencia (in, not in)

Además existe la comprobación de pertenencia ($x \in T$, **in**; $x \notin T$, **not in**) que técnicamente no es un operador y no se puede sobrecargar. Su precedencia es la misma que la de los operadores relacionales. Se puede utilizar con rangos o con subtipos.

-- *Supongamos que X es de tipo Integer* **if** X **in** Positive **then** -- *Positive es un subtipo de Integer* ... **if** X **not in** 4 .. 6 **then** ... **end if**; **end if**; **declare type** Dia_Semana **is** (Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo); **subtype** Dia_Laborable **is** Dia_Semana **range** Lunes .. Viernes; Hoy : Dia_Semana := Obtener_Dia; **begin if** Hoy **in** Dia_Laborable **then** -- *Dia_Laborable es un subtipo de Dia_Semana* Ir_Al_Trabajo; **if** Hoy **not in** Lunes .. Miercoles **then** Pensar_En_El_Fin_De_Semana; **end if**; **end if**; **end**;

66.4 Operadores lógicos de *corto-circuito*

Para los operadores lógicos existen versiones para minimizar las evaluaciones (*short-circuit evaluation*). Es decir, se evalúa primero el operando de la izquierda y después, sólo si es necesario para determinar el resultado, el de la derecha:

- Conjunción **and then**: no se evalúa la segunda expresión si la primera es falsa porque ya sabemos que el resultado será falso.
- Disyunción inclusiva **or else**: no se evalúa la segunda expresión si la primera es verdadera porque ya sabemos que el resultado será verdadero.

-- B / A > 3 *no se ejecutará si A es 0 lo que nos será útil para evitar -- un Constraint_Error --* **if** A /= 0 **and then** B / A > 3 **then** Put_Line ("B / A es mayor que 3"); **end if**;

66.5 Manual de referencia de Ada

- 4.5 Operators and Expression Evaluation

Capítulo 67

Programación en Ada/Pragmas

67.1 Descripción

Los **pragmas** son sentencias especiales que controlan el comportamiento del compilador, es decir son directivas de compilador. Tienen esta forma estándar:

pragma *Nombre* (*Lista_de_argumentos*);

La *Lista_de_argumentos* es opcional, no todos los pragmas necesitan argumentos.

67.2 Ejemplo

Imaginemos que deseamos hacer que una llamada a un subprograma se convierta en una expansión del código en el código objeto generado. Esto se suele hacer con subprogramas que son muy cortos, en los que se tarda más en realizar las operaciones necesarias en código máquina para llamar a la subrutina, que en realizar el código de la subrutina en sí. Para conseguir esta expansión del código se utiliza el pragma Inline.

function A_Pesetas (Euros : T_Euros) **return** T_Pesetas **is begin return** Euros * 166.386; **end** A_Pesetas; **pragma** Inline (A_Pesetas);

Siempre que se llame a A_Pesetas (Precio_en_Euros), el resultado en el código objeto sería equivalente a escribir Precio_en_Euros * 166.386 en lugar de la llamada.

67.3 Lista de pragmas definidos por el lenguaje

A continuación se presenta una tabla con los pragmas de Ada, sus argumentos, un enlace a la sección del Manual de referencia de Ada donde se define y una nota entre las que podemos encontrar:

Ada 2005 Este es un nuevo pragma que aparecerá en el estándar de Ada 2005.

Obsoleto Este es un pragma que se considera obsoleto pero se mantiene por compatibilidad. No es aconsejable usarlo en código nuevo.

67.3.1 A - H

67.3.2 I - O

67.3.3 P - R

67.3.4 S - Z

67.4 Lista de pragmas definidos por la implementación

La siguiente lista de pragmas no están disponibles en cualquier compilador, sólo en aquellos que han decidido implementarlos:

GNAT Éste es un pragma definido por la implementación del compilador GNAT.

67.4.1 A - C

- Abort_Defer (GNAT)
- Ada_83 (GNAT)
- Ada_95 (GNAT)
- Ada_05 (GNAT)
- Annotate (GNAT)
- Ast_Entry (GNAT)
- C_Pass_By_Copy (GNAT)
- Comment (GNAT)
- Common_Object (GNAT)
- Compile_Time_Warning (GNAT)
- Complex_Representation (GNAT)
- Component_Alignment (GNAT)
- Convention_Identifier (GNAT)
- CPP_Class (GNAT)

- CPP_Constructor (GNAT)
- CPP_Virtual (GNAT)
- CPP_Vtable (GNAT)

67.4.2 D - H

- Debug (GNAT)
- Elaboration_Checks (GNAT)
- Eliminate (GNAT)
- Export_Exception (GNAT)
- Export_Function (GNAT)
- Export_Object (GNAT)
- Export_Procedure (GNAT)
- Export_Value (GNAT)
- Export_Valued_Procedure (GNAT)
- Extend_System (GNAT)
- External (GNAT)
- External_Name_Casing (GNAT)
- Finalize_Storage_Only (GNAT)
- Float_Representation (GNAT)

67.4.3 I - L

- Ident (GNAT)
- Import_Exception (GNAT)
- Import_Function (GNAT)
- Import_Object (GNAT)
- Import_Procedure (GNAT)
- Import_Valued_Procedure (GNAT)
- Initialize_Scalars (GNAT)
- Inline_Always (GNAT)
- Inline_Generic (GNAT)
- Interface_Name (GNAT)
- Interrupt_State (GNAT)
- Keep_Names (GNAT)
- License (GNAT)
- Link_With (GNAT)
- Linker_Alias (GNAT)
- Linker_Section (GNAT)
- Long_Float (GNAT: OpenVMS)

67.4.4 M - S

- Machine_Attribute (GNAT)
- Main_Storage (GNAT)
- Obsolescent (GNAT)
- Passive (GNAT)
- Polling (GNAT)
- Profile_Warnings (GNAT)
- Propagate_Exceptions (GNAT)
- Psect_Object (GNAT)
- Pure_Function (GNAT)
- Restriction_Warnings (GNAT)
- Source_File_Name (GNAT)
- Source_File_Name_Project (GNAT)
- Source_Reference (GNAT)
- Stream_Convert (GNAT)
- Style_Checks (GNAT)
- Subtitle (GNAT)
- Suppress_All (GNAT)
- Suppress_Exception_Locations (GNAT)
- Suppress_Initialization (GNAT)

67.4.5 T - Z

- Task_Info (GNAT)
- Task_Name (GNAT)
- Task_Storage (GNAT)
- Thread_Body (GNAT)
- Time_Slice (GNAT)
- Title (GNAT)
- Unimplemented_Unit (GNAT)
- Universal_Data (GNAT)
- Unreferenced (GNAT)
- Unreserve_All_Interrupts (GNAT)
- Use_VADS_Size (GNAT)
- Validity_Checks (GNAT)
- Warnings (GNAT)
- Weak_External (GNAT)

67.5 Manual de referencia de Ada

67.5.1 Ada 95

- 2.8 Pragmas
- Annex L: (informative) Language-Defined Pragmas

67.5.2 Ada 2005

- 2.8 Pragmas
- Annex L (informative) Language-Defined Pragmas

Capítulo 68

Programación en Ada/Unidades predefinidas/Interfaces

Interfaces es el paquete padre de una serie de paquetes dedicados a facilitar el interfaz con otros lenguajes:

- Paquete Interfaces.C
  - Paquete Interfaces.C.Pointers
  - Paquete Interfaces.C.Strings
- Paquete Interfaces.Cobol
- Paquete Interfaces.Fortran

Además este paquete contiene varios tipos Integer_*n* y Unsigned_*n*, donde *n* es el número de bits que ocupa el tipo. Qué tipos concretos contiene depende del compilador y la arquitectura.

68.1 Manual de referencia de Ada

- B.2 The Package Interfaces

68.2 Text and image sources, contributors, and licenses

68.2.1 Text

- **Programación en Ada** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada?oldid=191971 *Colaboradores:* Javier Carro, ManuelGR, Almorca, Andres age, LadyInGrey, Chlewbot, Swazmo, ???? robot, Morza, MABot, Invadibot y Anónimos: 8
- **Programación en Ada/Introducción** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Introducción?oldid=158526 *Colaboradores:* ManuelGR, Morza y Anónimos: 2
- **Programación en Ada/Historia** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Historia?oldid=148657 *Colaboradores:* ManuelGR, Morza y Anónimos: 2
- **Programación en Ada/Manual de referencia** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Manual_de_referencia?oldid=124120 *Colaboradores:* ManuelGR, Morza y Anónimos: 1
- **Programación en Ada/Instalación** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Instalación?oldid=232070 *Colaboradores:* ManuelGR, Almorca, ???? robot, Morza, Rotlink y Anónimos: 3
- **Programación en Ada/Hola Mundo** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Hola_Mundo?oldid=129155 *Colaboradores:* ManuelGR, Ramac y Morza
- **Programación en Ada/Elementos del lenguaje** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Elementos_del_lenguaje?oldid=192912 *Colaboradores:* ManuelGR, Morza, Mecamático y Anónimos: 4
- **Programación en Ada/Tipos** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos?oldid=133834 *Colaboradores:* ManuelGR, Morza y CarsracBot
- **Programación en Ada/Tipos/Enteros** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Enteros?oldid=129158 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Tipos/Enumeraciones** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Enumeraciones?oldid=129159 *Colaboradores:* ManuelGR, ???? robot, Elbaygo, Morza y Anónimos: 3
- **Programación en Ada/Tipos/Coma flotante** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Coma_flotante?oldid=129160 *Colaboradores:* ManuelGR, ???? robot y Morza
- **Programación en Ada/Tipos/Coma fija** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Coma_fija?oldid=129161 *Colaboradores:* ManuelGR, ???? robot y Morza
- **Programación en Ada/Tipos/Arrays** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Arrays?oldid=168686 *Colaboradores:* ManuelGR, Chlewbot, Morza, Drinibot y Anónimos: 2
- **Programación en Ada/Tipos/Strings** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Strings?oldid=129163 *Colaboradores:* ManuelGR, ???? robot, Morza y Anónimos: 1
- **Programación en Ada/Tipos/Registros** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Registros?oldid=129164 *Colaboradores:* ManuelGR, Chlewbot, Morza y Anónimos: 1
- **Programación en Ada/Tipos/Registros discriminados** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Registros_discriminados?oldid=135301 *Colaboradores:* ManuelGR, Morza, CHV y Anónimos: 1
- **Programación en Ada/Tipos/Registros variantes** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Registros_variantes?oldid=208156 *Colaboradores:* ManuelGR, Morza y Anónimos: 2
- **Programación en Ada/Tipos/Punteros a objetos** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Punteros_a_objetos?oldid=133721 *Colaboradores:* ManuelGR, Morza y CarsracBot
- **Programación en Ada/Tipos/Punteros a subprogramas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos/Punteros_a_subprogramas?oldid=129171 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Subtipos** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Subtipos?oldid=194846 *Colaboradores:* ManuelGR, Jpcristian, Morza y Polyglottos
- **Programación en Ada/Tipos derivados** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos_derivados?oldid=129173 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Tipos etiquetados** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos_etiquetados?oldid=233771 *Colaboradores:* ManuelGR, ???? robot, Morza, Jcaraballo y Anónimos: 1
- **Programación en Ada/Diseño y programación de sistemas grandes** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Diseño_y_programación_de_sistemas_grandes?oldid=129175 *Colaboradores:* ManuelGR, Morza y Anónimos: 1
- **Programación en Ada/Paquetes** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Paquetes?oldid=167776 *Colaboradores:* ManuelGR, Chlewbot, Morza y Raulbcneo
- **Programación en Ada/Cláusula use** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Cláusula_use?oldid=129177 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Cláusula with** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Cláusula_with?oldid=129178 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Declaraciones** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Declaraciones?oldid=129182 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Ámbito** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Ámbito?oldid=129445 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Visibilidad** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Visibilidad?oldid=124158 *Colaboradores:* ManuelGR y Morza

- **Programación en Ada/Renombrado** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Renombrado?oldid=129446 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/La biblioteca** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/La_biblioteca?oldid=129447 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Unidades de biblioteca** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_de_biblioteca?oldid=129448 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Unidades hijas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_hijas?oldid=129449 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Subunidades** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Subunidades?oldid=129450 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Compilación separada y dependiente** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Compilación_separada_y_dependiente?oldid=129451 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Tipos abstractos de datos** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos_abstractos_de_datos?oldid=124165 *Colaboradores:* ManuelGR, Morza y Anónimos: 1
- **Programación en Ada/Tipos limitados** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tipos_limitados?oldid=124166 *Colaboradores:* ManuelGR, Morza y Anónimos: 1
- **Programación en Ada/Unidades genéricas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_genéricas?oldid=165255 *Colaboradores:* ManuelGR, Morza y Raulbcneo
- **Programación en Ada/Excepciones** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Excepciones?oldid=165437 *Colaboradores:* ManuelGR, Chlewbot, Morza, Raulbcneo y Anónimos: 2
- **Programación en Ada/Unidades predefinidas/Ada.Exceptions** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Ada.Exceptions?oldid=183569 *Colaboradores:* ManuelGR, Morza y Anónimos: 1
- **Programación en Ada/Concurrencia** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Concurrencia?oldid=129470 *Colaboradores:* ManuelGR, LadyInGrey, Morza y Anónimos: 1
- **Programación en Ada/Tareas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tareas?oldid=203043 *Colaboradores:* ManuelGR, Chlewbot, Morza, Igna y Anónimos: 2
- **Programación en Ada/Tareas/Sincronización mediante citas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Sincronización_mediante_citas?oldid=165447 *Colaboradores:* ManuelGR, Morza y Raulbcneo
- **Programación en Ada/Tareas/Aceptación de citas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Aceptación_de_citas?oldid=171175 *Colaboradores:* ManuelGR, Morza, Raulbcneo y Anónimos: 2
- **Programación en Ada/Tareas/Selección de citas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Selección_de_citas?oldid=178647 *Colaboradores:* ManuelGR, Morza y Anónimos: 2
- **Programación en Ada/Tareas/Llamadas a punto de entrada complejas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Llamadas_a_punto_de_entrada_complejas?oldid=130373 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Tareas/Dinámicas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Dinámicas?oldid=130448 *Colaboradores:* ManuelGR, BalDYxan, Morza y Anónimos: 1
- **Programación en Ada/Tareas/Dependencia** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Dependencia?oldid=130449 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Tareas/Ejemplos** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Tareas/Ejemplos?oldid=129149 *Colaboradores:* ManuelGR, BalDYxan y Morza
- **Programación en Ada/GLADE** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/GLADE?oldid=182094 *Colaboradores:* ManuelGR, Suruena, Morza, Luckas Blade, MABot, Invadibot y Anónimos: 1
- **Programación en Ada/Ada 2005** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Ada_2005?oldid=232054 *Colaboradores:* ManuelGR, Suruena, Morza y Rotlink
- **Programación en Ada/Unidades predefinidas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas?oldid=129584 *Colaboradores:* ManuelGR, □□□□ robot y Morza
- **Programación en Ada/Recursos en la Web** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Recursos_en_la_Web?oldid=231583 *Colaboradores:* ManuelGR, Morza, Raulbcneo, Invadibot y Rotlink
- **Programación en Ada/Subprogramas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Subprogramas?oldid=229745 *Colaboradores:* ManuelGR, Morza y Anónimos: 1
- **Programación en Ada/Unidades predefinidas/Standard** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Standard?oldid=129180 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Atributos** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Atributos?oldid=170753 *Colaboradores:* ManuelGR, Andres age, Jpcristian, Morza y Raulbcneo
- **Programación en Ada/Objetos** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Objetos?oldid=124139 *Colaboradores:* ManuelGR, Morza y Anónimos: 2
- **Programación en Ada/Entrada-salida** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Entrada-salida?oldid=134586 *Colaboradores:* ManuelGR, Jpcristian, Morza y Anónimos: 1
- **Programación en Ada/Unidades predefinidas/Ada.Text IO** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Ada.Text_IO?oldid=197603 *Colaboradores:* ManuelGR, Andres age, □□□□ robot, Russellhoff, Morza, Ralgisbot y Anónimos: 2

- **Programación en Ada/Unidades predefinidas/Ada.Float Text IO** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Ada.Float_Text_IO?oldid=124188 *Colaboradores:* ManuelGR, Morza y Anónimos: 1
- **Programación en Ada/Unidades predefinidas/Ada.Text IO.Editing** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Ada.Text_IO.Editing?oldid=129583 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Unidades predefinidas/Ada.Sequential IO** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Ada.Sequential_IO?oldid=197600 *Colaboradores:* ManuelGR, Andres age, Morza y Ralgisbot
- **Programación en Ada/Unidades predefinidas/Ada.Strings.Fixed** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Ada.Strings.Fixed?oldid=124201 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Unidades predefinidas/Ada.Strings.Unbounded** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Ada.Strings.Unbounded?oldid=124202 *Colaboradores:* ManuelGR, Morza y Anónimos: 1
- **Programación en Ada/Unidades predefinidas/Ada.Command Line** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Ada.Command_Line?oldid=124194 *Colaboradores:* ManuelGR y Morza
- **Programación en Ada/Operadores** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Operadores?oldid=124142 *Colaboradores:* ManuelGR, Chlewbot y Morza
- **Programación en Ada/Pragmas** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Pragmas?oldid=173216 *Colaboradores:* ManuelGR, Jpcristian, Morza y MABot
- **Programación en Ada/Unidades predefinidas/Interfaces** *Fuente:* http://es.wikibooks.org/wiki/Programación_en_Ada/Unidades_predefinidas/Interfaces?oldid=124193 *Colaboradores:* ManuelGR, Magister Mathematicae, Morza y Anónimos: 1

68.2.2 Images

- **Archivo:Ada_Lovelace.jpg** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/8/87/Ada_Lovelace.jpg *Licencia:* Public domain *Colaboradores:* File copied from english wikipedia at en:File:Ada_Lovelace.jpg *Artista original:* Margaret Sarah Carpenter
- **Archivo:Ada_Lovelace_1838.jpg** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/2/2e/Ada_Lovelace_1838.jpg *Licencia:* Public domain *Colaboradores:*

  From The Ada Picture Gallery.
  Evelyn Silva scanned this from a picture she found "in the trash" in Louisiana, USA, and submitted it to the Ada Picture Gallery in October 2000. She wrote: On the bottom of the picture it says "LONDON PUBLISHED NOV 1 1838 FOR THE PROPRIETORS, No 18 & 19 SOUTHAMPTON PLACE, EUSTON SQUARE, NEW ROAD". In the lower left corner it says "Printered by Mc Queen". On the lower right of the picture its "Engraved By W. H. Mote". On the left "Drawn by A.E. Chaton R.A.". There was also a page with a bio on it. This was not in a book when I found it, it was loose along with some other Ladies of the Queens court. So I don't have any other info on it. It is an original print from its time, not a reproduction.

  Artista original: William Henry Mote
- **Archivo:Dining_philosophers.png** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/6/6a/Dining_philosophers.png *Licencia:* CC-BY-SA-3.0 *Colaboradores:* Created by bdesham using OmniGraffle; post-processed in GraphicConverter. I used some other Commons images to make this one: *Artista original:* Benjamin D. Esham (bdesham)
- **Archivo:Estados_tareas.svg** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/c/cf/Estados_tareas.svg *Licencia:* CC-BY-SA-3.0 *Colaboradores:* ? *Artista original:* ?
- **Archivo:Evolution-tasks.png** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/9/93/Evolution-tasks.png *Licencia:* GPL *Colaboradores:* http://gnome.org/projects/evolution/download.shtml *Artista original:* Artwork by Tuomas Kuosmanen <tigert_at_ximian.com> and Jakub Steiner <jimmac_at_ximian.com>
- **Archivo:Help-books-aj.svg_aj_ash_01.svg** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/7/77/Help-books-aj.svg_aj_ash_01.svg *Licencia:* GPL *Colaboradores:* ? *Artista original:* ?
- **Archivo:Historie.png** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/d/db/Historie.png *Licencia:* CC-BY-SA-3.0 *Colaboradores:* ? *Artista original:* ?
- **Archivo:Info.svg** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/e/e1/Info.svg *Licencia:* Public domain *Colaboradores:* ? *Artista original:* ?
- **Archivo:Nuvola_apps_bookcase.svg** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/a/a5/Nuvola_apps_bookcase.svg *Licencia:* LGPL *Colaboradores:* ¡El código fuente de esta imagen SVG es <a data-x-rel='nofollow' class='external text' href='http://validator.w3.org/check?uri=http%3A%2F%2Fcommons.wikimedia.org%2Fwiki%2FSpecial%3AFilepath%2FNuvola_apps_bookcase.svg,,&,,ss=1#source'>válido.

  Artista original: Peter Kemp
- **Archivo:Tipos_Ada.png** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/9/91/Tipos_Ada.png *Licencia:* CC-BY-SA-3.0 *Colaboradores:* Trabajo propio *Artista original:* ManuelGR
- **Archivo:Wikipedia_logo_silver.png** *Fuente:* http://upload.wikimedia.org/wikipedia/commons/6/6e/Wikipedia_logo_silver.png *Licencia:* CC-BY-SA-3.0 *Colaboradores:* de:Image:Wikide.png (oder so) *Artista original:* User:Nohat

68.2.3 Content license

- Creative Commons Attribution-Share Alike 3.0