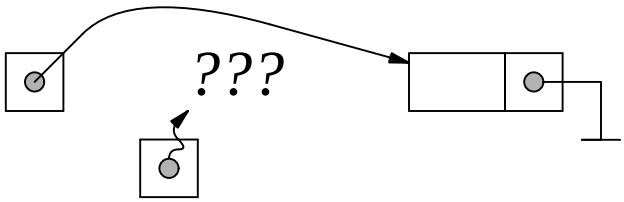
Curso Práctico de Programación



Usando Ada como Primer Lenguaje

Francisco J Ballesteros

A Esther.

Prefacio

Programar es una de las grandes cosas de la vida. Utilizar luego tus propios programas (como el sistema operativo que estoy utilizando para escribir esto) es una satisfacción aún mayor.

Aprender a programar no es una tarea trivial, aunque puede resultar fácil con la ayuda adecuada. En realidad programar es como cocinar. Hacen falta dos cosas para cocinar bien: Hacer muchos platos y ver muchas veces cómo se hacen distintos platos. Con el tiempo, los errores cometidos se solucionan y las técnicas que se ven en otros se aprenden.

La asignatura de *Fundamentos de Programación* para la que está escrito este libro tiene un fuerte handicap: Sólo existe un cuatrimestre para impartirla. A pesar de que hoy en día no hay mucha distinción entre Ingeniería de Telecomunicación e Ingeniería Informática en cuanto al tipo de trabajo que los titulados realizan en la industria, esto no parece haberse notado en la universidad al realizar los planes de estudio.

Este libro pretender **ir al grano** a la hora de enseñar cómo programar. No es una descripción teórica de la materia ni tampoco una colección de programas ya hechos para estudiar. El libro pretende enseñar, desde cero, a personas que no sepan programar cómo programar en un lenguaje tipo Ada. Se intenta no sólo describir los elementos básicos del lenguaje sino también **plasmar el proceso** mediante el cual se hacen los programas, en lugar de mostrarlos una vez concluidos. En pocas palabras, el texto trata de ser una versión escrita de las clases impartidas en la asignatura antes mencionada.

En ciertas ocasiones resulta necesario introducir ciertos conceptos antes de la realización de los programas que aparecen en el texto. Cuando es así, los conceptos están marcados en negrita. Si se utiliza el libro para repasar hay que comprobar que se conocen todos ellos antes de leer los programas o epígrafes que los siguen.

Los términos en **negrita** constituyen en realidad el vocabulario básico que hay que conocer para entender el resto del texto. Por cierto, utilizamos la *itálica* para referirnos a símbolos y palabras especiales. En los ejemplos utilizamos una fuente `monoespaciada` cuando nos referimos a texto de un programa o del ordenador en general. Siempre se usa ‘‘`;`’’ como símbolo del sistema (o *prompt*) en nuestros ejemplos. Todo el texto que escribimos nosotros en el ordenador se presenta en texto *`ligeramente inclinado`* o italizado. El texto escrito por el ordenador o por un programa se presenta siempre `sin inclinar`.

Hemos de decir que este libro no pretende enseñar a utilizar por completo el lenguaje Ada. Este es un lenguaje que no se distingue precisamente por ser pequeño e incluye numerosas características. Aunque hubiésemos preferido utilizar Pascal estándar, que es un lenguaje mucho más simple en tanto en cuanto no incluye tantas posibilidades, Ada es un lenguaje mucho más vivo. Por lo tanto el texto se limita a enseñar cómo utilizar un subconjunto de Ada similar a Pascal, lo que creemos es más conveniente para este curso que abrumar al alumno con aún más conceptos de los que ya se incluyen.

El texto no incluye estructuras de datos no lineales, y sólo enseña los rudimentos de la programación (la recursividad ni la mencionamos). Tras realizar este curso es preciso estudiar empleando un buen libro de estructuras de datos y de algoritmos, y leer cuanto más código mejor (siempre que dicho código esté escrito por buenos programadores y no por cualquiera que haya querido distribuir lo mejor que ha sabido hacer).

Los programas incluidos en el libro son manifiestamente mejorables. No obstante, entender las mejoras requiere más práctica que la que puede obtenerse en un cuatrimestre de programación. Hemos optado por intentar que se aprenda la mecánica habitual empleada para construir programas (si es que puede denominarse mecánica a algo que en realidad no tiene reglas y es más parecido a un arte). Todos los programas que se incluyen están hechos pensando en esto.

La bibliografía, lejos de estar completa, se ha intentado mantener lo más pequeña posible. Dicho de otro modo, la bibliografía indica aquellos libros que en nuestra opinión deberían leerse una vez terminado

el curso. Para una bibliografía completa siempre puede consultarse Google o utilizar la de los libros mencionados en la bibliografía.

Debo agradecer a Juan José Moreno y a José Manuel Burgos las clases de programación que me dieron durante la carrera (allá por el triásico superior). A José Manuel Burgos debo agradecerle también la ayuda prestada en la organización de este curso. Es uno de los mejores profesores que he conocido (¡Y no hay muchos!).

Si algunas partes del texto son difíciles de entender o si hay errores o sugerencias agradecería que se me indicara por correo electrónico.

Espero que todo esto resulte útil.

Francisco J. Ballesteros
nemo@lsub.org

Laboratorio de Sistemas,
Universidad Rey Juan Carlos de Madrid
Madrid, España
2009

Índice

Índice de programas

1 — Introducción a la programación

1.1. ¿Qué es programar?

Al contrario que otras máquinas, el ordenador puede hacer cualquier cosa que queramos si le damos las instrucciones oportunas. A darle estas instrucciones se le denomina **programar** el ordenador. Un ordenador es una máquina que tiene básicamente la estructura que puede verse en la figura 1.1. Posee algo de memoria para almacenar las instrucciones que le damos y tiene una unidad central para procesarlas (llamada CPU). En realidad, las instrucciones sirven para manipular información que también se guarda en la memoria del ordenador. A esta información la denominamos **datos** y a las instrucciones **programa**. Como la memoria no suele bastar y como su contenido se suele perder cuando eliminamos el suministro de energía, el ordenador también tiene un disco en el que puede almacenar tanta información (programas incluidos) como se desee de forma permanente.

Utilizando una analogía, podemos decir que un programa no es otra cosa que una receta de cocina, los datos no son otra cosa que los ingredientes para seguir una receta y la CPU es un pinche de cocina extremadamente tonto pero extremadamente obediente. Nosotros somos el Chef. Dependiendo de las órdenes que le demos al pinche podemos obtener o un exquisito postre o un armagedón culinario.

Es importante remarcar que la memoria del ordenador es volátil, esto es, se borra cada vez que apagamos el ordenador. La unidad central de proceso o **CPU** del ordenador es capaz de leer un programa ya almacenado en la memoria y de efectuar las acciones que dicho programa indica. Este programa puede manipular sólo datos almacenados también en la memoria. Esto no es un problema, puesto que existen programas en el ordenador que saben como escribir en la memoria los programas y datos que queremos ejecutar. A este último conjunto de programas se le suele llamar **sistema operativo**, puesto que es el sistema que nos deja operar con el ordenador.

El disco es otro tipo de almacenamiento para información similar a la memoria, aunque al contrario que ésta su contenido persiste mientras el ordenador está apagado. Su tamaño suele ser mucho mayor que el de la memoria del ordenador, por lo que se utiliza para mantener aquellos datos y programas que no se están utilizando en un momento determinado. Los elementos que guardan la información en el disco se denominan **ficheros** y no son otra cosa que nombres asociado a una serie de datos que guardamos en el disco (de ahí que se llamen ficheros, por analogía a los ficheros o archivos que empleamos en una oficina).

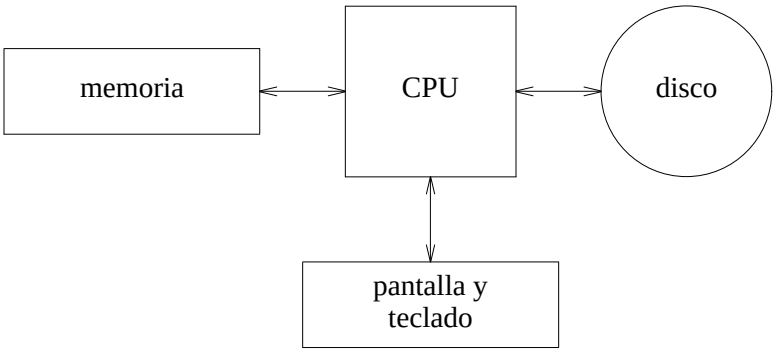


Figura 1.1: *Esquema de un ordenador.*

Tanto la memoria como el disco son capaces sólo de guardar información escrita como una serie de números. Estos números se escriben en base 2 o **binario**. Esto es, sólo podemos guardar unos y ceros en la memoria. Poder guardar sólo unos y ceros no es un problema. Con ellos podemos representar cualquier cosa. Imagina un objeto que puede ser persona, animal, planta o cosa. Si me

dejas hacerte suficientes preguntas que se respondan con ‘‘sí’’ o ‘‘no’’ es cuestión de hacer las preguntas adecuadas para saber en qué estas pensando. Si decimos que un 1 es un ‘‘sí’’ y un 0 es un ‘‘no’’ podríamos escribir tantos unos y ceros como respuestas afirmativas y negativas conduzcan a identificar el objeto en que se pensaba. Pues bien, toda esa secuencia de unos y ceros podrían ser la forma de representar este objeto en la memoria del ordenador.

Naturalmente la información no se guarda así. En lugar de eso se han buscado formas mejores de representarla. Por ejemplo, a cada una de las 256 operaciones básicas que podría saber hacer un ordenador le podemos asignar un número de 1 a 256. Si le damos uno de estos números a la CPU ella se ocupará de hacer la operación correspondiente. A lo mejor 15 significa sumar y 18 restar. Igualmente podemos hacer para almacenar texto escrito en el ordenador, podemos utilizar un número distinto para cada letra y guardarlos todos seguidos en la memoria según las letras que queramos escribir. Todo son números. Lo que significa cada número depende en realidad del programa que lo utiliza (y de la persona que escribe el programa).

Al conjunto de programas que hay en un ordenador se lo denomina **software** (literalmente *parte blanda*, que se refiere a los componentes no tangibles). Al conjunto físico de cables, plástico, metal y circuitos se lo denomina **hardware** (literalmente *parte dura* o tangible). Programar el ordenador consiste en introducir en el mismo un programa que efectúe las acciones que deseamos para resolver un problema. Dicho de otra forma, programar es construir software.

A pedirle al ordenador que siga los pasos que indica un programa se lo denomina **ejecutar** el programa. En realidad esto se lo tenemos que pedir al sistema operativo, que se ocupará de escribir o cargar en la memoria el programa que queremos ejecutar, tras lo cual la CPU hará el resto.

Antes de ver cómo programar necesitamos conocer nombres que se utilizan para medir cantidades de memoria. A una respuesta de tipo sí/no, esto es, a un dígito binario, se le denomina **bit** (por *binary digit*). Un número de 8 bits se denomina **byte**. Este tamaño tiene un nombre especial por su alta popularidad (por ejemplo, podemos utilizar un byte distinto para representar cada letra de un texto). A 1024 bytes se les suele llamar Kbyte (o **Kilobyte**). Se supone que en realidad deberíamos decir *Kibibyte* y escribirlo *Kibyte* pero nadie hace esto hoy día en la práctica. 1024 Kilobtyes hacen un Mbyte o **Megabyte** (debería decirse *Mebibyte*, aunque puede no sepan de lo que hablamos si utilizamos esta palabra). 1024 Mbytes constituyen un Gbyte o **Gigabyte**. Y así sucesivamente tenemos **Terabyte**, **Petabyte** y **Exabyte**.

La memoria de un ordenador suele ser de unos pocos Gigabytes. Los discos actuales suelen rondar el Terabyte. Todo esto nos importa para poder expresar cuánta memoria consume un programa o qué tamaño tiene el programa en si mismo, o cuánto almacenamiento necesitamos para los datos.

1.2. Programa para programar

En realidad programar es describir un plan con sumo detalle y empleando un lenguaje que luego el ordenador pueda entender. En este epígrafe vamos a ver cómo podemos proceder para construir programas. Lo podríamos denominar entonces un plan para programar, o un programa para programar. Lamentablemente el hardware es tan tonto que no sabría qué hacer con un plan expresado así, pero como no es el caso de nuestro cerebro, a nosotros este plan nos puede servir perfectamente. Los pasos que requiere realizar un programa son los siguientes:

1 **Definir el problema**. En primer lugar hay que tener muy claro qué es lo que se pretende resolver. Si no sabemos qué queremos hacer difícilmente sabremos cómo hacerlo. El problema es que, como vamos a tener que expresar con sumo detalle cómo hacerlo, también tenemos que saber con sumo detalle qué queremos hacer. A realizar este paso se le denomina **especificar** el problema.

2 **Diseñar un plan**. Una vez sabemos lo que queremos resolver necesitamos un plan. Este plan debería decirnos qué tenemos que hacer para resolver el problema.

3 **Implementar el plan**. Tener un plan no basta. Hay que llevarlo a cabo. Para llevarlo a cabo

hay que escribirlo en un lenguaje que entiende el ordenador.

4 **Probar el plan**. Una vez implementado, hay que probar el plan. En la mayoría de los casos nos habremos equivocado o habremos ignorado una parte importante del problema que queríamos resolver. Sólo cuando probamos nuestra implementación y vemos lo que hace ésta podemos ver si realmente estamos resolviendo el problema que queremos y cómo queremos.

5 **Depurar el plan**. Si nos hemos equivocado, lo que suele ser habitual, lo primero que necesitamos saber es en qué nos hemos equivocado. Posiblemente ejecutemos el plan varias veces para tratar de ver cuál es el problema. Una vez localizado éste, tenemos que volver al punto en el que nos hemos equivocado (cualquiera de los mencionados antes).

Veamos en qué consiste cada uno de estos pasos. Pero antes, un aviso importante: **estos pasos no son estancos**. No se hace uno por completo y luego se pasa al siguiente. Si queremos tener éxito hay que abordar el problema por partes y efectuar todos estos pasos para cada pequeña parte del problema original. De otro modo no obtendremos nada útil, por muchos pasos que sigamos.

Esto último es tan importante que es la diferencia entre realizar programas que funcionan y construir auténticas atrocidades. **Las cosas se hacen poco a poco y por pasos, haciéndolo mal la primera vez y mejorando el resultado hasta que sea satisfactorio.**

Dijimos que programar es como cocinar y que un programa es como una receta de cocina. Piensa cómo llegan los grandes Chefs a conseguir sus recetas. Ninguno de ellos se sienta a trazar un plan maestro extremadamente detallado sin haber siquiera probado qué tal sabe una salsa. Tal vez, el chef se ocupe primero de hacer el sofrito, ignorando mientras tanto el resto de la receta. Quizás, se ocupe después de cómo asar la carne y de qué pieza asar (tal vez ignorando el sofrito). En ambos casos hará muchos de ellos hasta quedar contento con el resultado. Probará añadiendo una especia, quitando otra, etc. Programar es así.

Se programa **refinando progresivamente** el programa inicial, que posiblemente no cumple prácticamente nada de lo que tiene que cumplir para solucionar el problema, pero que al menos hace algo. En cada paso se produce un prototipo del programa final. En principio, el programa se termina cuando se dice basta en este proceso de mejora a fuerza de iterar los pasos mencionados anteriormente.

1.3. Refinamiento del programa para programar

Con lo dicho antes puede tenerse una idea de qué proceso hay que seguir para construir programas. Pero es mejor refinar nuestra descripción de este proceso, para que se entienda mejor y para conseguir el vocabulario necesario para entenderse con otros programadores. ¿Se ve cómo estamos refinando nuestro programa para programar? Igual hay que hacer con los programas.

Empecemos por definir el problema. ¿Especificación? ¿Qué especificación? Aunque en muchos textos se sugiere construir laboriosas y detalladas listas de requisitos que debe cumplir el software a construir, o seguir determinado esquema formal, nada de esto suele funcionar bien para la resolución de problemas abordables por programas que no sean grandes proyectos colectivos de software. Incluso en este caso, es mucho mejor una descripción precisa aunque sea más informal que una descripción formal y farragosa que puede contener aún más errores que el programa que intenta especificar.

Para programar individualmente, o empleando equipos pequeños, es suficiente tener claro cuál es el problema que se quiere resolver. Puede ayudar escribir una descripción informal del problema, para poder acudir a ella ante cualquier duda. Es bueno incluir todo aquello que queramos que el programa pueda hacer y tal vez mencionar todo lo que no necesitamos que haga. Por lo demás nos podemos olvidar de este punto.

Lo que importa en realidad para poder trabajar junto con otras personas son los **interfaces**. Esto es, qué tiene que ofrecerle nuestra parte del programa a los demás y qué necesita nuestra parte del programa de los demás. Si esto se hace bien el resto viene solo. En otro caso es mejor

cambiar de trabajo (aunque sea temporalmente).

Una vez especificado el problema necesitamos un plan detallado para resolverlo. Este plan es el **algoritmo** para resolver el problema. Un algoritmo es secuencia detallada de acciones o alguna otra forma de definir cómo se calcula la solución del problema. Por poner un ejemplo, un algoritmo para freír un huevo podría ser el que sigue:

```
Tomar sarten
Si falta aceite entonces
       Tomar aceite y
       Poner aceite en sarten
Y despues seguir con...
Poner sarten en fogon
Encender fogon
Mientras aceite no caliente
       no hacer nada
Y despues seguir con...
Tomar huevo
Partir huevo
Echar interior del huevo a sarten
Tirar cascara
Mientras huevo crudo
       mover sarten (para que no se pegue)
Y despues seguir con...
Retirar interior del huevo de sarten
Tomar plato
Poner interior del huevo en plato
Apagar fogon
```

A este lenguaje utilizado para redactar informalmente un algoritmo se le suele denominar **pseudocódigo**, dado que en realidad este texto intenta ser una especie de código para un programa. Pero no es un programa: no es algo que podamos darle a un ordenador para que siga sus indicaciones, esto es, para que lo ejecute. Se puede decir que el pseudocódigo es un lenguaje intermedio entre el lenguaje natural (el humano) y el tipo de lenguajes en los que se escriben los programas. Luego volveremos sobre este punto.

En realidad un ordenador no puede ejecutar los algoritmos tal y como los puede escribir un humano. Es preciso escribir un programa (un texto, guardado en un fichero) empleando un lenguaje con sintaxis muy estricta que luego puede traducirse automáticamente y sin intervención humana al lenguaje que en realidad habla el ordenador (binario, como ya dijimos).

Al lenguaje que se usa para escribir un programa se le denomina, sorprendentemente, **lenguaje de programación**. El lenguaje que de verdad entiende el ordenador se denomina **código máquina** y es un lenguaje numérico. Un ejemplo de un programa escrito en un lenguaje de programación podría ser el siguiente:

```
procedure FreirHuevo is
begin
       Get(sarten);
       if Cantidad(aceite) < Minima then
              Get(aceite);
              PonerEn(sarten, aceite);
       end if;
       PonerEn(fogon, sarten);
       Encender(fogon);
       while not EstaCaliente(aceite) loop
              null;
       end loop;
       Get(huevo);
       Partir(huevo, cascara, interior);
       PonerEn(sarten, interior);
       Delete(cascara);
       while Crudo(huevo) loop
              Mover(sarten);-- para que no se pegue
       end loop;
       Get(plato);
       QuitarDe(sarten, interior);
       PonerEn(plato, interior);
       Apagar(fogon);
end;
```

Al texto de un programa (como por ejemplo este que hemos visto) se le denomina **código fuente** del programa. Programar es en realidad escribir código fuente para un programa que resuelve un problema. A cada construcción (o frase) escrita en un lenguaje de programación se la denomina **sentencia**. Por ejemplo,

```
Delete(cascara);
```

es una sentencia.

Para introducir el programa en el ordenador lo escribimos en un fichero empleando un **editor**. Un editor es un programa que permite editar texto y guardarlo en un fichero. Como puede verse, todo son programas en este mundo. Es importante utilizar un editor de texto (como *TextEdit*) y no un procesador de textos o para escribir documentos (como *Word*) puesto que estos últimos están más preocupados por el aspecto del texto que por el texto en sí. Si utilizamos un procesador de textos para escribir el programa, el fichero contendrá muchas más cosas además del texto (negritas, estilo, etc.) y no podremos traducir el texto a binario puesto que el programa que hace la traducción no lo entenderá.

A la acción de escribir el texto o código de un programa se la suele denominar también **codificar** el algoritmo o **implementar** (o realizar) el algoritmo.

Una vez codificado, tenemos que traducir el texto a binario. Para esto empleamos un programa que realiza la traducción. Este programa se denomina **compilador**; se dice que compila el lenguaje de programación en que está escrito su código fuente. Por tanto, **compilar** es traducir el programa al lenguaje del ordenador empleando otro programa para ello, este último ya disponible en binario.

Un compilador toma un fichero como datos de entrada, escrito en el lenguaje que sabe compilar. A este fichero se le denomina **fichero fuente** del programa. Como resultado de su ejecución dicho compilador genera otro fichero ya en el lenguaje del ordenador denominado **fichero objeto**. El fichero objeto ya está redactado en binario, pero no tiene todo el programa. Tan sólo tiene las partes del programa que hemos escrito en el fichero fuente. Normalmente hay muchos trozos del programa que tomamos prestados del lenguaje de programación que utilizamos y que se encuentran en otros ficheros objeto que ya están instalados en el ordenador, o que hemos compilado anteriormente.

Piensa que, en el caso de las recetas de cocina, nuestro chef puede tener un pinche que sólo habla chino mandarín. Para hacer la receta el chef, que es de Cádiz, escribe en español diferentes textos (cómo hacer el sofrito, cómo el asado, etc.). Cada texto lo manda traducir por separado a chino mandarín (obteniendo el objeto del sofrito, del asado, etc.). Luego hay que reunir todos lo objetos en una única receta en chino mandarín y dársela al pinche.

Una vez tenemos nuestros ficheros objetos, y los que hemos tomados prestados, otro programa denominado **enlazador** se ocupa de juntar o enlazar todos esos ficheros en un sólo binario que pueda ejecutarse. De hecho, suele haber muchos ficheros objeto preparados para que los usemos en nuestros programas en las llamadas **librerías** (o bibliotecas) del sistema. Estas librerías que son ficheros que contienen código ya compilado y listo para pasar al enlazador. Todo este proceso está representado esquemáticamente en la figura 1.2.

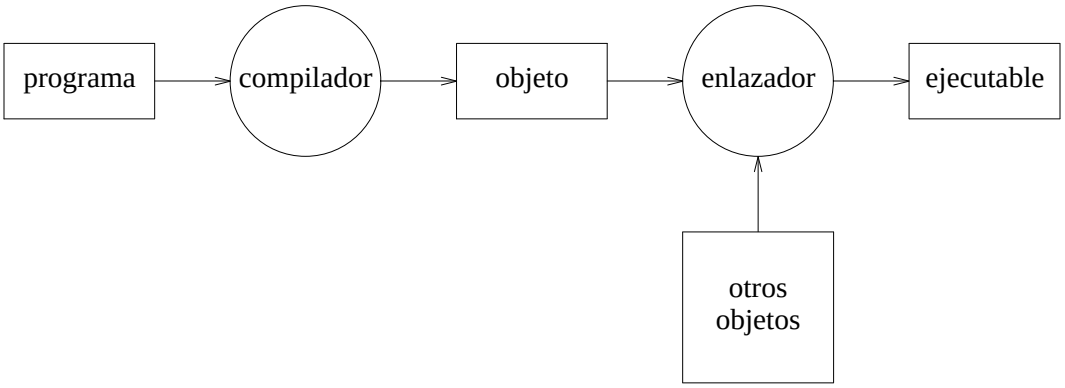


Figura 1.2: *El compilador genera ficheros objeto que se enlazan y forman un ejecutable.*

Un fichero ejecutable puede tener el aspecto que sigue, si utilizamos base 16 para escribir los números que lo forman (llamada **hexadecimal** habitualmente):

```
0000000  4e6f726d 616c6d65 6e746520 73652075
0000010  74696c69 7a612065 6c207465 636c6164
0000020  6f20792f 6f20656c 20726174 c3b36e20
0000030  70617261 20696e64 69636172 6c652061
0000040  6c0a7369 7374656d 61207175 6520676f
 ...
```

Claramente no queremos escribir esto directamente. El ejecutable contiene **instrucciones** que son **palabras** (números) que corresponden a órdenes concretas para la CPU. Los datos que maneja el programa en la memoria del ordenador tienen el mismo aspecto. Son palabras, esto es, números. Por eso se suele denominar binarios a los ficheros objeto y a los ficheros ejecutables, por que contienen números en base 2 (en binario) que es el lenguaje que en realidad entiende el ordenador.

Una vez tenemos un ejecutable, lo primero es ejecutar el programa. Normalmente se utiliza el teclado y/o el ratón para indicarle al sistema que gobierna el ordenador (al sistema operativo) que ejecute el programa. La ejecución de un programa consiste en la **carga** del mismo en la memoria del ordenador y en conseguir que la CPU comience a ejecutar las instrucciones que componen el programa (almacenadas ya en la memoria).

En general, todo programa parte de unos datos de entrada (normalmente un fichero normal o uno especial, como puede ser el teclado del ordenador) y durante su ejecución genera unos datos de salida (normalmente otro fichero, que puede ser normal o uno especial, como puede ser la pantalla del ordenador). En cualquier caso, a los datos los denominamos **entrada** del programa y a los resultados producidos los denominamos **salida** del programa. El esquema lo podemos ver en la figura 1.3.

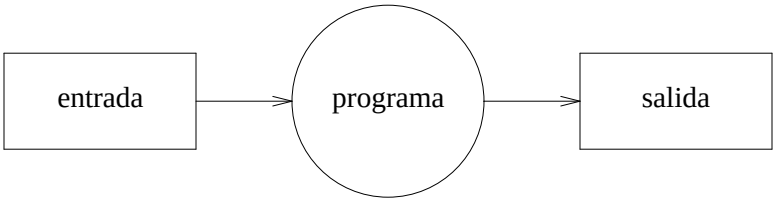


Figura 1.3: *Un programa toma datos de la entrada y tras procesarlos produce resultados en la salida.*

Las primeras veces que ejecutamos el programa estamos interesados en **probarlo** (comprobar que el resultado es el esperado). En muchos casos cometeremos errores al escribir un programa. Algunos de ellos son errores sintácticos (o de otro tipo) que puede detectar el compilador. Estos son **errores de compilación** que impiden al compilador generar un ejecutable. Si cometemos este tipo de errores deberemos arreglar el programa y volver a intentar su compilación. En otros casos, un error en el programa consiste en ejecutar instrucciones erróneas (por ejemplo, dividir por cero) que provocarán que el programa tenga problemas en su ejecución. Naturalmente el programa hace lo que se le dice que haga, pero podemos no haber tenido en cuenta ciertos casos que conducen a un error. Estos son **errores de ejecución** que provocan que el programa deje de ejecutar cuando se producen, dado que el ordenador no sabría qué hacer a continuación del error.

Además tenemos los llamados **errores lógicos**, que consisten en que el programa ejecuta correctamente pero no produce los resultados que esperamos (a lo mejor hemos puesto el huevo primero y la sartén después y aunque esperábamos comer nos toca limpiar en ayunas).

El propósito de probar el programa es intentar descubrir nosotros todos los errores posibles antes de que los sufran otros (y causen un perjuicio mayor). Si un programa no se prueba, intentando romperlo por todos los medios posibles, lo más seguro es que el programa no funcione correctamente. Por cierto, sea cual sea el tipo de error, se le suele llamar **bug**. Siempre que alguien habla de un ‘‘bug’’ en un programa se refiere a un error en el mismo.

En cualquier caso, ante un error, es preciso ver a qué se debe dicho error y luego arreglarlo en el programa. Las personas que no saben programar tienden a intentar ocultar o arreglar el error antes siquiera de saber a qué se debe. Esto es la mejor receta para el desastre. Si no te quieres pasar horas y horas persiguiendo errores lo mejor es que no lo hagas.

Una vez se sabe a qué se debe el error se puede cambiar el programa para solucionarlo. Y por supuesto, una vez arreglado, hay que ver si realmente está arreglado ejecutando de nuevo el programa para probarlo de nuevo. A esta fase del desarrollo de programas se la denomina fase de pruebas. Es importante comprender la importancia de esta fase y entender que forma parte del desarrollo del programa.

Pero recuerda que una vez has probado el programa seguro que has aprendido algo más sobre el problema y posiblemente quieras cambiar la especificación del mismo. Esto hace que de nuevo vuelvas a empezar a redefinir el problema, rehacer un poco el diseño del algoritmo, cambiar la implementación para que corresponda al nuevo plan y probarla de nuevo. No hay otra forma de hacerlo.

1.4. Algoritmos

Normalmente un algoritmo (y por tanto un programa) suele detallarse empleando tres tipos de construcciones: secuencias de acciones, selecciones de acciones e iteraciones de acciones. La idea es emplear sólo estas tres construcciones para detallar cualquier algoritmo. Si se hace así, resultará fácil realizar el programa correspondiente y evitar parte de los errores.

Una **secuencia** es simplemente una lista de acciones que han de efectuarse una tras otra. Por ejemplo, en nuestro algoritmo para freír un huevo,

```
Tomar huevo
Partir huevo
Echar interior del huevo a sarten
Tirar cascara
```

es una secuencia. Hasta que no se termina una acción no comienza la siguiente, por lo que la ejecución de las mismas se produce según se muestra en la figura 1.4.

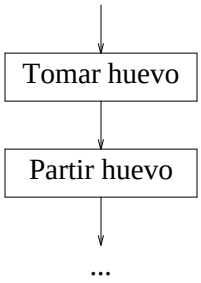


Figura 1.4: *Una secuencia ejecuta acciones una tras otra hasta concluir.*

Aquí lo importante es que **sistemáticamente se ejecuta una acción tras otra**. Siempre se comienza la ejecución por el principio (la parte superior de la figura) y se termina por el final (la parte inferior de la figura). En ningún momento se comienza directamente por una acción intermedia ni se salta desde una acción intermedia en la secuencia hasta alguna otra acción que no sea la siguiente.

La importancia de hacer esto así es que de no hacerlo resultaría tremendamente difícil comprender lo que realmente hace el algoritmo y sería muy fácil cometer errores. Errores que luego tendríamos que depurar (lo cual es justo lo que nunca queremos hacer, dado que depurar es una tarea dolorosa).

A este conjunto de acciones que comienzan a ejecutar en un único punto (arriba) y terminan en otro único punto (abajo) lo denominamos **bloque** de sentencias.

Otro elemento que utilizamos para construir algoritmos es la **selección**. Una selección es una construcción que efectúa una acción u otra dependiendo de que una condición se cumpla o no se cumpla. Por ejemplo, en nuestro algoritmo para freír un huevo,

```
Si falta aceite entonces
       Tomar aceite
       Poner aceite
Y despues seguir con...
```

es una selección. Aquí el orden de ejecución de acciones sería como indica la figura 1.5.

La condición es una pregunta que debe poder responderse con un sí o un no, esto es, debe ser una proposición verdadera o falsa. De este modo, esta construcción ejecuta una de las dos ramas. Por ejemplo, en la figura, o se ejecutan las acciones de la izquierda o se ejecuta la acción de la derecha. A cada una de estas partes (izquierda o derecha) se la denomina **rama** de la selección. Así, tenemos una rama para el caso en que la condición es cierta y otra para el caso en que la condición es falsa.

Igual que sucede con la secuencia, la ejecución siempre comienza por el principio (la parte superior de la figura) y termina justo detrás de la selección (la parte inferior en la figura). No es posible comenzar a ejecutar arbitrariamente. Tanto si se cumple la condición como si no se cumple (en cualquiera de las dos alternativas) podemos ejecutar una o más acciones (o quizá ninguna) en la rama correspondiente.

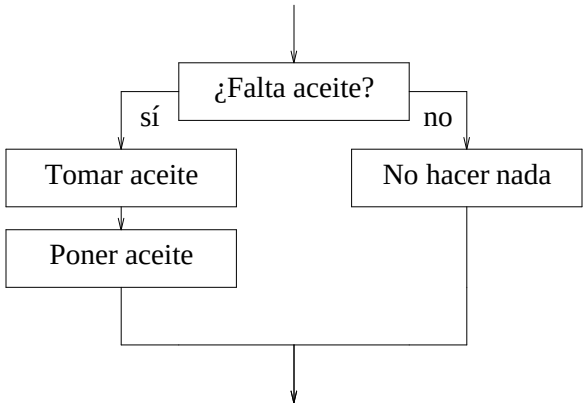


Figura 1.5: *Una selección ejecuta una u otra acción según una condición sea cierta o falsa.*

Podemos utilizar una selección en un algoritmo en cualquier sitio donde podamos utilizar una sentencia (dado que el flujo u orden en que se ejecutan las cosas empieza siempre por arriba y termina siempre por abajo). Además, como se ha podido ver, en cada rama podemos tener un bloque de sentencias (y no una única sentencia). Estos bloques pueden a su vez incluir selecciones, naturalmente.

Sólo hay otra construcción disponible para escribir algoritmos y programas: la **iteración**. Una iteración es una construcción que repite un bloque (una secuencia, una selección o una iteración) cierto número de veces. Por ejemplo, nuestro algoritmo culinario incluye esta iteración:

```
Mientras huevo crudo
       mover sarten
Y despues seguir con...
```

El orden de ejecución de esta construcción puede verse en la figura 1.6.

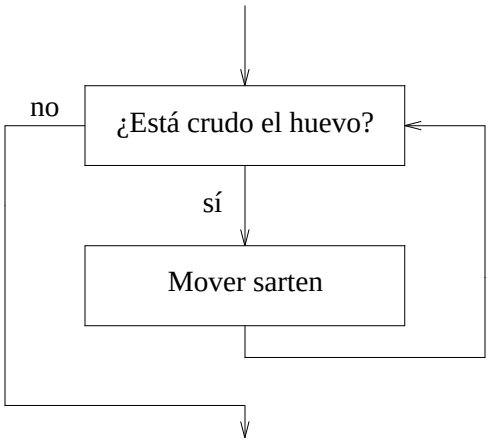


Figura 1.6: *Una iteración repite algunas otras sentencias mientras sea preciso.*

Como puede verse, también se comienza a ejecutar por un único punto al principio y se termina la ejecución en un único punto al final. En todos los casos (secuencia, selección o iteración) lo hacemos así. Esto hace que el algoritmo sea **estructurado** y sea posible seguir con facilidad la secuencia de ejecución de acciones. Aún mas, hace que sea posible cambiar con facilidad el algoritmo. Cuando hagamos un cambio podemos pensar sólo en la parte que cambiamos, dado que no es posible saltar arbitrariamente hacia el interior de la parte que cambiamos. Siempre se comienza por un punto, se hace algo y se termina en otro punto.

Cuando ejecutemos el programa que corresponde al algoritmo de interés, tendremos un orden de ejecución de acciones que podemos imaginar como una mano que señala a la acción en curso y que avanza conforme continúa la ejecución del programa. A esto lo denominamos **flujo de control** del programa. El único propósito de la CPU es en realidad dar vida al flujo de control de un programa. Naturalmente no hay ‘‘manos’’ en la CPU. En su lugar, la CPU mantiene la cuenta de qué instrucción es la siguiente a ejecutar (mediante el llamado **contador de programa**). Pero podemos olvidarnos de esto por el momento.

De igual modo que estructuramos el flujo de control empleando secuencias, selecciones e iteraciones, también es posible agrupar y estructurar los datos tal y como como veremos en el resto del curso. Citando el título de un excelente libro de programación:

Algoritmos + Estructuras de datos = Programas

El mencionado libro [1] es un muy buen libro para leer una vez completado este curso.

1.5. Programas en Ada

Ada es un lenguaje de programación que puede utilizarse para escribir programas de ordenador, los cuales sirven por supuesto para manipular datos en el ordenador.

Ada, como cualquier lenguaje de programación, manipula entidades **abstractas**, esto es, que no existen en realidad más que en el lenguaje. Quizá sorprendentemente a estas entidades se las denomina datos abstractos.

Podemos tener lenguajes más abstractos (más cercanos a nosotros) y lenguajes más cercanos al ordenador, o menos abstractos. Por eso hablamos del **nivel de abstracción** del lenguaje. En Ada el nivel de abstracción del lenguaje está lejos del nivel empleado por el ordenador. Por eso decimos que Ada es un **lenguaje de alto nivel** (de abstracción, se entiende).

Esto es fácil de entender. Ada habla de conceptos (abstracciones) y el ordenador sólo entiende de operaciones elementales y números (elementos concretos y no muy abstractos). Por ejemplo, Ada puede hablar de símbolos que representan días de la semana aunque el ordenador sólo manipule números enteros (que pueden utilizarse para identificar días de la semana). Sucede lo mismo en cualquier otro lenguaje.

Antes de seguir... ¡Que no cunda el pánico! El propósito de este epígrafe es tomar contacto con el lenguaje. No hay que preocuparse de entender el código (los programas) que aparece después. Tan sólo hay que empezar a ver el paisaje para que luego las cosas nos resulten familiares. Dicho de otro modo, vamos a hacer una visita turística a un programa.

Un programa Ada tal y como los que vamos a realizar durante este curso presenta una estructura muy definida y emplea una sintaxis muy rígida. Lo que sigue es un ejemplo de programa. Lo vamos a utilizar para hacer una disección (como si de una rana se tratase) para que se vean las partes que tiene. Pero primero el código:

programa.adb

```
1     --
2     -- Programa en Ada.
3     -- Autor: aturing
4     --

6     --
7     -- Entorno. Paquetes que usamos.
8     --
9     with Ada.Text_IO;
10    use Ada.Text_IO;
```

```
12    --
13    -- cabecera del programa.
14    --
15    procedure Programa is

17          --
18          -- Tipos y Constantes
19          --

21          -- Dia de la Semana
22          type TipoDiaSem is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
23          -- π
24          Pi : constant Float := 3.1415926;

26          --
27          -- Paquetes que "creamos".
28          --

30          package Float_InOut is new Float_IO(Float);
31          use Float_InOut;

33          --
34          -- Funciones y procedimientos
35          --

37          -- Longitud de la circunferencia para un circulo de radio r
38          function LogitudCircunferencia(r: Float) return Float is
39          begin
40               return 2.0 * Pi * r;
41          end;

43          --
44          -- Constantes de pruebas.
45          --
46          RadioPrueba: constant Float := 2.0;
47          Prueba1: constant Float := LogitudCirculo(RadioPrueba);

49          --
50          --  Declaraciones del programa principal
51          --
52          saludo: String := "Hola que tal.";

54    --
55    -- Cuerpo del programa principal.
56    --
57    begin
58          -- Pruebas
59          Put(Prueba1);

61          -- Programa
62          Put(saludo);
63          New_Line;
64    end;
—
```

Lo primero que hay que decir es que este programa debe estar guardado en un fichero en el ordenador. De otro modo no sería útil en absoluto (salvo que tengamos que empapelar algo). En este libro escribimos el nombre de cada fichero en un pequeño rectángulo en negrita al principio

de cada programa. Por ejemplo, el programa que hemos visto se guardaría en un fichero llamado `programa.adb`.

Nótese que los números de línea mostrados a la izquierda de cada línea no son parte del programa (no hay que escribirlos). Los hemos incluido al presentar los programas para que sea más fácil referirse a una línea o líneas concretas.

Este fichero fuente se utilizará como entrada para un compilador de Ada, con el propósito de obtener un ejecutable como salida del compilador, tras la fase de enlazado, y poder ejecutar en el ordenador nuestro programa. Recuerda, todos los programas son lo mismo: utilizan una entrada, hacen algo y producen una salida.

Todo el texto mostrado constituye el **código fuente** del programa. Como puede verse, es texto escrito directamente sin emplear ni negrita, ni itálica, ni ningún otro elemento de estilo. Hay que tener en cuenta que lo único que le importa a Ada es el texto que hemos escrito y no cómo lo hemos escrito. Por eso dijimos que hay que emplear para escribir programas editores de texto y no programas como *Word*, que están en realidad más preocupados por cómo queda el texto que por otra cosa (y por consiguiente incluyen en el fichero información sobre cómo hay que presentar el texto; lo que confundiría al compilador).

Comencemos con la anatomía del programa. Llama la atención que hay líneas en el programa que comienzan por ‘‘`--`’’. Estas líneas son **comentarios** y son líneas que el compilador ignora por completo. Esto es, todo el texto desde un ‘‘`--`’’ hasta el final de esa línea es texto que podríamos borrar del programa si queremos y el programa seguiría siendo el mismo. Los comentarios están pensados para los humanos que leen el programa (con frecuencia los mismos humanos que han escrito el programa).

Es importante que un programa sea **legible** fácilmente por humanos. Esto es aún más importante que otros criterios tales como la eficiencia a la hora de utilizar los recursos del ordenador; los comentarios ayudan. Por ejemplo, estas líneas del programa anterior son un comentario:

```
1     --
2     -- Programa en Ada.
3     -- Autor: aturing
4     --
```

Este comentario ayuda a que veamos para qué sirve el programa. A lo mejor leyendo ese comentario nos basta y nos ahorramos leer el programa entero. Nótese que si queremos escribir un comentario de varias líneas todas ellas han de comenzar por ‘‘`--`’’. Esa es la sintaxis que Ada impone para los comentarios (otros lenguajes lo hacen de otro modo).

Otra cosa que podemos ver mirando el programa anterior es que un programa tiene varias secciones bien definidas, que habitualmente vamos a tener en todos los programas. En este curso nuestros programas tendrán siempre estas secciones en el orden que mostramos, aunque algunas de ellas podrán omitirse si están vacías en un programa dado. En nuestro programa hemos incluido comentarios al principio de cada sección para explicar el propósito de la misma. Vamos a verlas en orden.

La primera sección ya la conocemos. Es un comentario mostrando el propósito del programa y el autor del mismo. Por ejemplo:

```
--
-- Programa en Ada de ejemplo.
-- Autor: Sheldom
--
```

Algo informal y breve que diga qué es esta cosa que estamos mirando es en realidad el comentario más útil que podríamos hacer.

A continuación podemos ver otra sección que se utiliza para conectar el programa con su **entorno**, esto es, con el resto del software en el ordenador:

```
6     --
7     -- Entorno. Paquetes que usamos.
8     --
9     with Ada.Text_IO;
10    use Ada.Text_IO;
```

En este caso estaríamos diciendo que este programa tiene intención (en principio) de utilizar librerías ya existentes en el sistema para leer y escribir texto. Ada llama a las librerías **paquetes**, como puedes ver por el comentario.

Podemos ver que en esta y otras secciones hay palabras que están sujetas a la sintaxis del lenguaje. Por ejemplo, para indicar que utilizamos la librería `Ada.Text_IO` hay que escribir exactamente

```
with Ada.Text_IO;
use Ada.Text_IO;
```

y no es posible omitir nada. No podemos quitar ninguno de los ‘‘`;`’’, por ejemplo. A lo largo del programa emplearemos palabras que tienen un significado especial. Aquí, la palabra *with* es una de esas palabras; decimos que es una **palabra reservada** o *palabra clave* del lenguaje (En inglés decimos *keyword*). Esto quiere decir que es una palabra que forma parte del lenguaje y se utiliza para un propósito muy concreto. No podemos usar la palabra *with* para ningún otro propósito que para decir que el programa desea utilizar un paquete o librería, por ejemplo.

En un programa tambien utilizamos otras palabras para referirnos a elementos en el lenguaje o a elementos en el mundo (según lo ve el programa, claro). Por ejemplo, *Ada* y *Text_IO* son palabras que, juntas en `Ada.Text_IO`, nombran un paquete (de software) de Ada. Estas palabras son **identificadores**, dado que identifican elementos en nuestro programa. En general, tendremos algunos identificadores ya definidos que podremos utilizar y también podremos definir otros nosotros.

Los identificadores son palabras que deben comenzar por una letra y sólo pueden utilizar letras o números (pero teniendo en cuenta que a ‘‘`_`’’ se le considera una letra). No pueden comenzar con otros símbolos tales como puntos, comas, etc. Piensa que el compilador que debe traducir este lenguaje a binario es un programa, y por lo tanto no es muy listo. Si no le ayudamos empleando una sintaxis y normas férreas no sería posible implementar un compilador para el lenguaje y tendríamos que hacer la traducción a mano.

Ahora que los mencionamos, a los símbolos que utilizamos para escribir (letras, números, signos de puntuación) los denominamos **caracteres**. Incluso un espacio en blanco es un carácter.

Los siguientes nombres son identificadores correctos en Ada:

```
Ada
Put_Line
Put0
getThisOrThat
X32__
```

Pero estos otros **no** lo son:

```
0x10
Ada.Text_IO
Fecha del mes
0punto
```

Si no ves por qué, lee otra vez las normas para escribir identificadores que acabamos de exponer y recuerda que el espacio en blanco es un carácter como otro cualquiera. El tercer identificador de este ejemplo no es válido puesto que utiliza caracteres blancos (esto es, espacios en blanco) como parte del nombre.

`Ada.Text_IO` tampoco es un identificador válido ¿O creías que sí? En realidad en nuestro programa este nombre está compuesto de *dos* identificadores: *Ada* y *Text_IO*. Los paquetes de Ada (como *Ada*) pueden tener dentro otros paquetes (como *Text_IO*). Eso es lo que significa `Ada.Text_IO`, que es un nombre de paquete que emplea dos identificadores para nombrar el paquete *Text_IO* dentro del paquete *Ada*. Todo esto no importa mucho ahora, salvo por entender lo que es un identificador y cómo inventar nuevos identificadores (nuevos nombres) que sean válidos en Ada.

Por cierto, Ada no distingue entre mayúsculas y minúsculas en cuanto a identificadores se refiere (palabras clave inclusive). Por lo tanto *Programa*, *programa*, *PROGRAMA* y *ProGraMa* son todos el mismo identificador. Deberás recordar que esto es así en Ada, pero hemos de decir que en otros lenguajes sí se distingue entre mayúsculas y minúsculas y los ejemplos anteriores serían identificadores diferentes.

Continuando con nuestra descripción de las secciones de un programa la siguiente que encontramos es realmente importante: la cabecera del programa.

```
12    --
13    -- cabecera del programa.
14    --
15    procedure Programa is
```

Aquí, las palabras *procedure* e *is* son palabras reservadas que se utilizan para indicar que *Programa* es a partir de este momento el nombre (el identificador) del programa que estamos escribiendo. Dado que el propósito de este programa es mostrar un programa en Ada, *Programa* es un buen nombre. Pensemos que este programa no tiene otra utilidad que la de ser un programa. Por lo demás, es un programa que no sirve para gran cosa. Si hiciésemos un programa para escribir la tabla de multiplicar del 7 un buen nombre sería *TablaDel7* y en cambio resultaría que llamarlo *Programa* sería una pésima elección. ¡Por supuesto que es un programa! El nombre debería decir algo que no sepamos sobre el programa. En realidad, debería decirlo todo en una sola palabra.

Normalmente, el nombre del programa se hace coincidir con el nombre del fichero que contiene el programa. Ahora bien, el nombre del fichero fuente en Ada ha de terminar en ‘‘`.adb`’’. Por ejemplo, nuestro programa de ejemplo para esta disección debería estar en un fichero llamado ‘‘`programa.adb`’’.

Tras la cabecera del programa tenemos otra sección dedicada a definir nuevos elementos en el mundo donde vive nuestro programa. Esta sección se denomina sección de constantes y tipos (que ya veremos lo que son). Por ejemplo, en nuestro programa estamos definiendo los días de la semana para que el programa pueda manipular días y también la constante π.

```
17          --
18          -- Tipos y Constantes
19          --

21          -- Dia de la Semana
22          type TipoDiaSem is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
23          -- π
24          Pi : constant Float := 3.1415926;
```

Suele ser útil indicar en un comentario antes de cada definición el propósito de la misma. Aunque suele ser aún mejor que la definición resulte obvia. Dados los nombres empleados en las estas definiciones podríamos haber escrito:

```
type TipoDiaSem is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
Pi : constant Float := 3.1415926;
```

Esto se entiende perfectamente y es más fácil de leer. Aquí, *type*, *is*, *constant* y *Float* son palabras reservadas del lenguaje y no podemos emplear ninguna otra en su lugar. En cambio,

TipoDiaSem, *Lun*, *Pi*, etc. son identificadores y podríamos haber empleado otros cualesquiera.

Podemos ver además que en un programa hay símbolos que se representan a si mismos. Por ejemplo, 3.1415926 es un número real concreto y *Mar* es un día de la semana concreto. Estos símbolos se denominan **literales** y se utilizan para referirse a ciertos elementos literalmente. Por ejemplo, quizá resulte sorprendente que el literal 3.1415926 se refiere al número 3.1415926 de forma literal. Por eso lo denominamos literal.

Aunque Ada no distingue mayúsculas de minúsculas (salvo cuando están escritas entrecomilladas), se suelen respetar normas respecto a cuándo emplear mayúsculas y cuándo minúsculas a la hora de escribir los identificadores y palabras reservadas en un programa. Hacerlo así tiene la utilidad de que basta ver un identificador escrito para saber no sólo a qué objeto se refiere, sino también de que tipo de objeto se trata.

Por ejemplo, las palabras reservadas las vamos a escribir siempre en minúscula. En cuanto veamos escrito *type*, podemos sospechar que es una palabra reservada que hace algo muy concreto en este lenguaje. Igualmente, los nombres de constantes los vamos a escribir siempre comenzando por mayúscula (con el resto en minúsculas). Así pues, *Pi* parece ser una constante (y como tal sería un error intentar siquiera cambiar su valor). Los nombres de tipos los capitalizamos igual que los de constantes. Por ejemplo, *TipoDiaSem* en nuestro caso. Eso si, haremos siempre que el nombre empiece por *Tipo* (para saber que hablamos de un tipo; aunque ahora mismo no sepamos de lo que hablamos).

La siguiente sección que encontramos en cualquier programa se utiliza para crear versiones particulares de paquetes que están escritos para funcionar en casos generales. Ya veremos lo que quiere decir esto conforme los utilicemos durante el curso. En nuestro programa esta sección es:

```
26          --
27          -- Paquetes que "creamos".
28          --

30          package Float_InOut is new Float_IO(Float);
31          use Float_InOut;
```

Tenemos que saber que normalmente se denomina **instanciar** a crear algo; y a ese algo se le denomina **instancia**. Por ejemplo, este programa está instanciando el paquete *Float_IO*. Si no se entiende nada de esto no pasa nada. Lo único que importa por el momento es ver las distintas partes del programa y entender el vocabulario que vamos a emplear durante el resto del curso.

La siguiente sección en nuestro programa es la de funciones y procedimientos. Esta sección define pequeños programas auxiliares (o **subprogramas**) que sirven para calcular algo necesario para el programa que nos ocupa. En nuestro ejemplo, esta sección es como sigue:

```
33          --
34          -- Funciones y procedimientos
35          --

37          -- Longitud de la circunferencia para un circulo de radio r
38          function LogitudCircunferencia(r: Float) return Float is
39          begin
40               return 2.0 * Pi * r;
41          end;
```

Este fragmento de programa define la función *LogitudCircunferencia*, similar a la función matemática empleada para la calcular la longitud de la circunferencia correspondiente a un círculo de radio *r*. Es aconsejable incluir un breve comentario antes de la función indicando el propósito de la misma. Aunque, como dijimos, si los nombres se escogen adecuadamente posiblemente el comentario resulta innecesario. Así, una función llamada *LongitudCircunferencia* seguramente no haga otra cosa que calcular la longitud de una circunferencia.

Esta función recibe un número real y devuelve un número real, similar a lo que sucede con una función matemática con domino en $\mathbb{R}$ e imagen en $\mathbb{R}$. El resultado para *LogitudCircunferencia(r)* sería el valor de $2\pi r$. Luego volveremos sobre esto, aunque puede verse que

```
40              return 2.0 * Pi * r;
```

es la sentencia que indica cómo calcular el valor resultante de la función.

La siguiente sección es la denominada de constantes de pruebas. En ella definimos constantes que emplean los subprogramas y elementos definidos anteriormente para ejercitarlos y poder ver si funcionan correctamente o no antes de utilizarlos de verdad en el programa.

```
43          --
44          -- Constantes de pruebas.
45          --
46          RadioPrueba: constant Float := 2.0;
47          Prueba1: constant Float := LogitudCirculo(RadioPrueba);
```

Puede verse que tiene un aspecto similar a la declaración de la constante π que vimos antes. La principal diferencia es que estas definiciones son totalmente arbitrarias y no tienen más utilidad que la de utilizarlas para probar el programa. Por eso es bueno mantenerlas en una sección separada del resto del programa.

Es extremadamente importante probar todas las funciones y subprogramas antes de utilizarlos. De otro modo no podemos estar seguros de que funcionen bien. Es más, en otro caso podemos estar realmente seguros de que *no* funcionan bien.

A continuación nos encontramos con la sección de declaraciones para el programa principal. Esta sección define elementos que necesitamos en el programa que sigue. Su interés es definir objetos llamados **variables** (similares a las variables en matemáticas) que vamos a emplear en el programa.

```
49          --
50          --  Declaraciones del programa principal
51          --
52          saludo: String := "Hola que tal.";
```

Y por último, nos falta lo más importante: el programa. En nuestro caso este es como sigue:

```
54      --
55      -- Cuerpo del programa principal.
56      --
57      begin
58          -- Pruebas
59          Put(Prueba1);

61          -- Programa
62          Put(saludo);
63          New_Line;
64      end;
—
```

Podemos ver que normalmente el programa incluye sentencias para efectuar pruebas (que luego desparecerán una vez el programa funcione) y sentencias para realizar las acciones que constituyen el algoritmo que queremos emplear. A esta parte la denominamos **cuerpo** del programa. También se la conoce como **programa principal**, en consideración a que los subprogramas que también forman parte del fuente son en realidad programas. En cualquier caso, cuando ejecutemos el programa este empezará a realizar las acciones o sentencias indicadas en su cuerpo. Eso sí, tras efectuar las definiciones de constantes encontradas antes.

En este programa utilizamos dos sentencias que resultarán muy útiles a lo largo de todo el curso. La sentencia *Put* escribe en la pantalla algo que le indicamos entre paréntesis. La sentencia *New_Line* hace que el programa continúe escribiendo en otra línea distinta. Cuando ejecutemos este programa, Ada realizará cada una de las acciones indicadas por cada sentencia del programa principal. Este programa es de hecho una secuencia de acciones, similar a las que hemos mencionado antes al hablar de algoritmos.

Hemos podido ver a lo largo de todo este epígrafe que ciertas declaraciones y sentencias están escritas con sangrado, llamado también **tabulación**, de tal forma que están escritas más a la derecha que los elementos que las rodean. Esto se hace para indicar que éstas se consideran dentro de la estructura definida por dichos elementos.

Por ejemplo, las sentencias del programa principal están dentro del bloque de sentencias escrito entre *begin* y *end*:

```
begin
        ...
end;
```

Por eso están sangradas (a la derecha). Están escritas comenzando por un tabulador (que se escribe pulsando la tecla del tabulador en el teclado, indicada con una flecha a la derecha en la parte izquierda del teclado). Así, en cuanto vemos el programa podemos saber a simple vista que ciertas cosas forman parte de otras. Esto es muy importante puesto nos permite **olvidarnos** de todo lo que esté dentro de otra cosa (a no ser que esa cosa sea justo lo que nos interese).

Como hemos podido ver, la estructura de un programa viene impuesta por la sintaxis de Ada y siempre es:

```
procedure Programa is
      ...
begin
      ...
end;
```

Todas las definiciones o declaraciones hechas entre *is* y *begin* están sangradas o tabuladas; lo mismo que sucede con el cuerpo del programa (las sentencias entre *begin* y *end*).

Un último recordatorio: los signos de puntuación son importantes en un programa, dado que el compilador es otro programa y por tanto no muy listo. El compilador depende de los signos de puntuación para reconocer la sintaxis del programa.

Por ahora diremos sólo dos cosas respecto a esto: (1) en Ada *todas las sentencias terminan con un* ‘‘`;`’’ y (2) signos tales como *paréntesis y comillas deben de ir por parejas y bien agrupados*.

Esto no quiere decir que todas las líneas del programa terminen en un punto y coma. Quiere decir que las sentencias (y las declaraciones) han de hacerlo. Por ejemplo, *begin* no tiene nunca un punto y coma detrás. En cambio, *end* sí lo tiene dado que

```
begin
      ...
end;
```

se considera una sentencia, aunque compuesta de sentencias entre el *begin* y el *end*. Esto ya lo hemos visto en realidad. Estas sentencias son un bloque de sentencias, escritas en Ada, pero un bloque al fin y al cabo. Este bloque ejecuta una secuencia de acciones (igual que vimos para los algoritmos en pseudocódigo).

Por ejemplo,

```
Put("Hola mundo")
```

es una sentencia incorrecta en Ada dado que le falta un ‘‘`;`’’ al final. Igualmente,

```
Put "Hola mundo");
```

es incorrecta dado que le falta un paréntesis. Del mismo modo,

```
(3 + ) (3 - 2 )
```

es una expresión incorrecta dado que el signo ‘‘+’’ requiere de un segundo valor para sumar, que no encuentra antes de cerrar la primera pareja de paréntesis. Todo esto resultará natural conforme leamos y escribamos programas y no merece la pena decir más por ahora.

1.6. ¡Hola π!

Para terminar este capítulo vamos a ver un famoso programa en Ada denominado ‘‘Hola π’’. Este programa resultará útil para los problemas que siguen y para el próximo capítulo.

holapi.adb

```
1     --
2     -- Saludar al numero π
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure HolaPi is

10          Pi: constant Float := 3.1415926;

12          -- necesario para escribir π
13          package FloatIO is new Float_IO(Float);
14          use FloatIO;

16    begin
17          Put("Hola ");
18          Put(Pi);
19          Put("!");
20          New_Line;
21    end;
```

Este programa define una constante llamada *Pi* en la línea 10 y luego las sentencias del programa principal se ocupan de escribir un mensaje de saludo para dicha constante. Por ahora obviaremos el resto de detalles de este programa.

Para compilar este programa nosotros daremos la orden ‘‘`ada`’’ al sistema operativo indicando el nombre del fichero que contiene el código fuente. Suele decirse que ‘‘ejecutamos el comando *ada*’’. En tu caso es muy posible que baste con utilizar el ratón si utilizas Windows o que tengas que dar la orden o ejecutar el comando *gnatmake* si utilizas Linux. Para ejecutar el programa utilizamos el nombre del fichero ejecutable producido por el compilador como orden para el sistema operativo. En tu caso puede ser que baste de nuevo con utilizar el ratón si utilizas Windows o Linux.

```
; ada holapi.adb
; holapi
Hola  3.14159E+00!
```

En adelante mostraremos la salida de todos nuestros programas de este modo. Lo que el programa ha escrito en este caso es

```
Hola  3.14159E+00!
```

y el resto ha sido sencillamente lo que hemos tenido que escribir nosotros en nuestro sistema para ejecutar el programa (pero eso depende del ordenador que utilices).

En adelante utilizamos siempre ‘‘`;`’’ como símbolo del sistema (o *prompt*) en nuestros ejemplos. Todo el texto que escribimos nosotros en el ordenador se presenta en texto *`ligeramente inclinado`* o italizado. El texto escrito por el ordenador o por un programa se presenta siempre `sin inclinar`.

Problemas

1 Compila, enlaza y ejecuta el programa ‘‘hola π’’ cuyo único propósito es escribir un saludo. Busca el código del programa en Internet o tómalo de la página de recursos de la asignatura.

2 Borra un punto y coma del programa. Compílalo y explica lo que pasa. Soluciona el problema.

3 Borra un paréntesis del programa. Compílalo y explica lo que pasa. Soluciona el problema.

4 Elimina la última línea del programa. Compílalo y explica lo que pasa. Soluciona el problema.

5 Elimina las dos primeras líneas del programa. Compílalo y explica lo que pasa. Soluciona el problema.

6 Escribe tú el programa desde el principio, compílalo, enlázalo y ejecútalo.

7 Cambia el programa para que salude al mundo entero y no sólo al número π.

2 — Elementos básicos

2.1. ¿Por dónde empezamos?

En el capítulo anterior hemos tomado contacto con la programación y con el lenguaje Ada. Por el momento, una buena forma de proceder es tomar un programa ya hecho (como por ejemplo el ‘‘Hola π’’ del capítulo anterior) y cambiar sólo las partes de ese programa que nos interesen. De ese modo podremos ejecutar nuestros propios programas sin necesidad de, por el momento, saber cómo escribirlos enteros por nuestros propios medios.

Recuerda que programar es como cocinar (o como conducir). Sólo se puede aprender a base de práctica y a base de ver cómo practican otros. No es posible aprenderlo realmente sólo con leer libros (aunque puede ayudar). Si no tienes un compilador de Ada cerca, busca uno. Úsalo para probar por ti mismo cada una de las cosas que veas durante este curso. La única forma de aprender a usarlas es usándolas.

2.2. Conjuntos y elementos

El lenguaje Ada permite básicamente manipular datos. Eso si, estos datos son entidades abstractas y están alejadas de lo que en realidad entiende el ordenador. Programar en Ada consiste en aprender a definir y manipular estas entidades abstractas, lo cual es más sencillo de lo que parece.

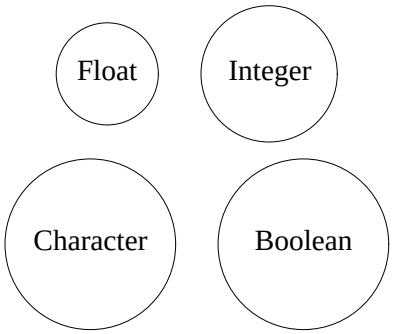


Figura 2.1: *El mundo según Ada: existen enteros, caracteres, reales y valores de verdad.*

¿Cómo es el mundo para Ada? En realidad es tan sencillo como muestra la figura 2.1. Para Ada, en el mundo existen entidades (cosas) que pueden ser números reales, números enteros, caracteres o valores de verdad. ¡Y no hay nada más! (por el momento).

Esto quiere decir que en Ada tenemos estos cuatro conjuntos diferentes (números reales, números enteros, caracteres y valores de verdad). Cada uno de ellos consta de elementos homogéneos entre sí. El conjunto de los caracteres contiene, sorprendentemente, caracteres; igual sucede con el conjunto de los números reales, que contiene números reales.

Cada conjunto tiene unas operaciones bien definidas para trabajar sobre sus elementos. Por ejemplo, podemos sumar números enteros entre sí para obtener otros números enteros. Igualmente, podemos sumar números reales entre sí para obtener otros números reales. Pero *no podemos mezclar números enteros con números reales*.

Quizá sorprenda esto, pero así son las cosas. Si queremos sumar un número entero a un número real tendremos que conseguir un número real que tenga el mismo valor que el número entero, y luego sumarlo. La razón por la que esto es así es intentar evitar que, por accidente, hagamos cosas tales como sumar peras con manzanas; o contamos peras o contamos manzanas o

contamos piezas de fruta. Pero no podemos mezclarlo todo a voluntad.

A cada conjunto se lo denomina **tipo de datos** y, naturalmente, a los elementos de cada conjunto se los denomina **datos**. Resumiendo:

- Un tipo de datos es un conjunto de elementos con unas operaciones bien definidas para datos pertenecientes a ese tipo.
- No es posible combinar datos de distinto tipo entre sí.

Debido a esto se dice que Ada es un lenguaje **fuertemente tipado**. Hay otros lenguajes de programación que permiten combinar entre sí elementos de distintos tipos, pero esto da lugar a muchos errores a la hora de programar.

Ada utiliza la palabra reservada *Integer* para denominar al tipo de datos (conjunto) de los números enteros. El conjunto de los reales está representado en Ada por el tipo de datos *Float*. De igual manera, tenemos el conjunto de los valores de verdad, ‘‘cierto’’ y ‘‘falso’’, representado en Ada por el tipo de datos *Boolean*. Este último sólo tiene dos elementos: *True* representa en Ada al valor de verdad ‘‘cierto’’ y *False* representa al valor de verdad ‘‘falso’’. Tenemos también un tipo de datos para el conjunto de caracteres disponible, llamado *Character*.

Así, el literal 3 pertenece al tipo de datos *Integer* dado que es un número entero y sólo puede combinarse normalmente con otros elementos también del tipo *Integer*. La siguiente expresión en Ada intenta sumar dos y tres:

```
2 + 3.0                ¡incorrecto!
```

Un entero sólo puede sumarse con otro entero; nunca con un número real. Por tanto, dicha expresión es incorrecta y dará lugar a un error durante la compilación del programa que la incluya. Lo hemos intentado (cambiando un poco el programa de saludo a π) y esto es lo que ha pasado:

```
; ada malo.adb
malo.adb:20:15: invalid operand types for operator "+"
malo.adb:20:15: left operand has type universal integer
malo.adb:20:15: right operand has type universal real
ada: "malo.adb" compilation error
```

El compilador ha escrito un mensaje para informar de un error, intentando describirlo de la forma más precisa que ha podido, y no ha hecho nada más. Esto es, **no ha compilado el programa**. ¡No tenemos un fichero ejecutable! El programa contenía simplemente la sentencia

```
Put(2 + 3.0);
```

y, como ya sabemos, estas sentencias no compilan dado que mezclan elementos de distintos tipos.

La razón es que debe existir **compatibilidad de tipos** entre los elementos que aparecen en una expresión. Esto es, podemos escribir

```
2 + 3
```

y también

```
2.0 + 4.3
```

pero no expresiones que mezclen valores de tipo *Integer* y valores de tipo *Float* como en la expresión incorrecta mostrada arriba.

Los literales de tipo *Float* pueden también escribirse en notación exponencial, expresados como un número multiplicado por 10 elevado a una potencia dada. Por ejemplo,

```
3.2E-3
```

es en realidad $3.2x10^{-3}$, esto es: 0.0032.

En aquellas ocasiones en que necesitamos sumar un entero a un número real, podemos realizar una **conversión de tipo** sobre uno de los valores para conseguir un valor equivalente en el tipo de datos que queremos utilizar. Por ejemplo,

```
Float(3)
```

es una expresión cuyo valor es en realidad 3.0, de tipo *Float*. Las conversiones de tipo tienen siempre este aspecto, pero no siempre se permiten. Por ejemplo, no podemos convertir un carácter a un número entero, igual que no tendría sentido convertir una pera en una manzana o el plomo en oro (bueno, esto último tal vez si).

2.3. Operaciones

Como estamos viendo, cada tipo de datos tiene un nombre, por ejemplo *Integer*, y define un conjunto de elementos de dicho tipo. Además, cada tipo de datos tiene unas operaciones básicas que nos permiten manipular datos de dicho tipo. El reflejo en Ada de estas operaciones básicas (que son en realidad operaciones matemáticas) son los llamados **operadores**. Por ejemplo, ‘‘+’’ es un operador para el tipo *Integer* y podemos utilizarlo para sumar números enteros en Ada.

Un operador permite operar sobre uno o más elementos de un tipo de datos determinado. A estos elementos se les denomina **operandos**. Algunos operadores se escriben entre los operandos (por ejemplo, la suma) y se les denomina **infijos**. Otros operadores adoptan otras formas. Por ejemplo, el cambio de signo se escribe con un ‘‘-’’ escrito antes del objeto sobre el que opera; por eso se denomina operador **prefijo**. Otros operadores adoptan el aspecto de funciones, como por ejemplo el operador de conversión de tipo que hemos visto antes para convertir un 3 en un valor de tipo *Float*.

Los tipos de datos numéricos admiten los siguientes operadores:

- Suma: `3 + 4`
- Resta: `4.0 - 7.2`
- Multiplicación: `2 * 2`
- División: `7 / 3`
- Exponencial: `2 ** 3` (Esto es, 2^3)
- Valor absoluto: `Abs(3)`
- Conversión a otro tipo numérico: `Float(3)`, `Integer(3.5)`

En realidad, tenemos un juego de estos operadores para los enteros y otro distinto para los reales, aunque se llamen igual. Así, la división es concretamente una división entera (parte entera del cociente) cuando los números son enteros y es una división real en otro caso.

El tipo *Integer* dispone además de los siguientes operadores:

- Resto: `3 rem 2`
- Módulo: `5 mod 3`

El resto corresponde al resto de la división entera. Y el módulo a la operación matemática del mismo nombre. Normalmente se utilizan siempre con números positivos, en cuyo caso ambas operaciones producen el mismo valor.

La lista de operadores la hemos incluido aquí más como una referencia que como otra cosa. Podemos olvidarnos de ella. A medida que los uses los recordarás de forma natural.

2.4. Expresiones

Empleando los operadores que hemos descrito es posible escribir expresiones más complejas que calculen valores de forma similar a como hacemos en matemáticas. El problema de la escritura de expresiones radica en que para Ada el programa es simplemente una única secuencia de

caracteres. Así es como ve Ada el programa para saludar la número π:

```
--\n-- Saludar al numero π\n--\n\nwith Ada.Text_IO;\nuse Ada.Text_IO;
\n\nprocedure HolaPi is\n\n→Pi: constant Float := 3.1415926;\n\n→--
 necesario para escribir π\n→package FloatIO is new Float_IO(Float);
\n→use FloatIO;\n\nbegin\n→Put("Hola ");\n→Put(Pi);\n→Put("!");\n→
New_Line;\nend;\n
```

Lo ve como si todas las líneas del fichero estuvieran escritas en una única línea extremadamente larga. Esto quiere decir que no podemos escribir un 3 sobre el signo de dividir y una suma bajo el mismo para expresar que el denominador es una suma. Por ejemplo, para dividir 3 por 2 tenemos que escribir

```
3 / 2
```

pero no podemos escribir $\frac{3}{2}$.

Igualmente, para sumar los números del 1 al 5 y dividir el resultado por 3 no podemos utilizar

$$\frac{1+2+3+4+5}{3}$$

En su lugar, es preciso escribir:

```
(1 + 2 + 3 + 4 + 5) / 3
```

Los paréntesis son necesarios para agrupar la expresión que suma los números del uno al cinco, de tal forma todas las sumas estén en el numerador.

Lo que pasa es que los operadores se evalúan empleando un orden determinado. Por eso se dice que algunos tienen **precedencia** sobre otros (que el lenguaje los escoge y los evalúa antes que estos otros). Por ejemplo, la división y la multiplicación tienen precedencia sobre las sumas y restas. Esto es, las divisiones y las multiplicaciones en una expresión se calculan antes que las sumas y las restas. En el ejemplo anterior, de no utilizar los paréntesis estaríamos calculando en realidad:

```
1 + 2 + 3 + 4 + 5 / 3
```

Esto es,

```
1 + 2 + 3 + 4 + (5 / 3)
```

O lo que es lo mismo:

$$1+2+3+4+\frac{5}{3}$$

La exponenciación tiene precedencia sobre la multiplicación y la división. Por tanto,

```
2 * 3 ** 2
```

está calculando

```
2 * (3 ** 2)
```

y no

```
(2 * 3) ** 2
```

Cuando existan dudas sobre la precedencia conviene utilizar paréntesis para agrupar las expresiones según deseemos.

Una nota importante es que *se debe utilizar el espacio en blanco y los signos de puntuación (paréntesis en este caso) para hacer más legible la expresión.* Como puede verse en las expresiones de ejemplo, los espacios están escritos de tal modo que resulta más fácil leerlas. Normalmente, se escriben antes y después del operador, pero nunca después de un paréntesis abierto o antes de uno cerrado.

2.5. Otros tipos de datos

El tipo de datos *Character* representa un carácter del juego de caracteres disponibles. Estos son algunos ejemplos de literales de tipo carácter:

```
'A'
'0'
' '
```

El primero representa a la letra ‘‘A’’, el segundo al dígito ‘‘0’’ y el último al espacio en blanco. Todos ellos son de tipo *Character*. Es importante ver que `'0'` es un carácter y `0` es un entero. El primero se usa al manipular texto y el segundo al manipular números. ¡No tienen que ver entre si! De hecho... ¡No pueden operarse entre sí dado que son de distinto tipo!

Un carácter se almacena en el ordenador como un número cuyo valor representa el carácter en cuestión. Inicialmente se utilizaba el código **ASCII** para representar los caracteres aunque hoy día son populares otros códigos como **UTF**. Una codificación de caracteres no es más que una tabla en la que se asigna un valor numérico a cada carácter que se quiera representar (como ya sabemos, los ordenadores sólo saben manejar números). En la mayoría de los casos, las letras sin acentuar, dígitos y signos comunes de puntuación empleados en Inglés están disponibles en el juego de caracteres ASCII. Suele ser buena idea *no* emplear acentos en los identificadores que empleamos en los programas por esta razón. Aunque Ada lo permite, no es así en otros lenguajes y no todo el software de ordenador (editores por ejemplo) se comporta igual con caracteres acentuados.

Otro tipo de datos importante es el llamado booleano, *Boolean* en Ada. Este tipo representa valores de verdad: ‘‘Cierto’’ y ‘‘Falso’’. Está compuesto sólo por el conjunto de elementos *True* y *False*, que corresponden a ‘‘Cierto’’ y ‘‘Falso’’. ¿Recuerdas la selección en los algoritmos?

Este tipo es realmente útil dado que se emplea para expresar condiciones que pueden cumplirse o no, y para hacer que los programas ejecuten unas sentencias u otras dependiendo de una condición dada.

Los operadores disponibles para este tipo son los existentes en el llamado **álgebra de Boole** (de ahí el nombre del tipo).

- Negación: `not`
- Conjunción: `and`
- Disyunción: `or`
- Disyunción exclusiva: `xor`

Sus nombres son pintorescos pero es muy sencillo hacerse con ellos. Sabemos que *True* representa algo que es cierto y *False* representa algo que es falso. Luego si algo no es falso, es que es cierto. Y si algo no es cierto, es que es falso. Por tanto:

```
not True = False
not False = True
```

Sigamos con la conjunción. Si algo que nos dicen es en parte verdad y en parte mentira... ¿Nos están mintiendo?. Si queremos ver si dos cosas son conjuntamente verdad utilizamos la conjunción. Sólo es cierta una conjunción de cosas ciertas:

```
False and False = False
False and True = False
True and False = False
True and True = True
```

Sólo resulta ser verdad *True and True*. ¡El resto de combinaciones mienten!

Ahora a por la disyunción. Esta nos dice que alguna de dos cosas es cierta (o tal vez las dos). Esto es:

```
False or False = False
False or True = True
True or False = True
True or True = True
```

En cuanto uno de los operandos sea cierto la disyunción es cierta. Hay otro tipo de disyunción que es exclusiva y sólo es cierta cuando un único operando sea cierto. Se utiliza para el caso típico en que se quiere expresar que o bien sucede una cosa o bien sucede otra (pero no las dos a la vez):

```
False xor False = False
False xor True = True
True xor False = True
True xor True = False
```

Con algún ejemplo más todo esto resultará trivial. Sea *pollofrito* un valor que representa que un pollo está frito y *pollocrudo* un valor que representa que un pollo está crudo. Esto es, si el pollo está frito tendremos que

```
pollofrito = True
pollocrudo = False
```

Eso sí, cuando el pollo esté crudo lo que ocurrirá será mas bien

```
pollofrito = False
pollocrudo = True
```

Luego ahora sabemos que

```
pollofrito xor pollocrudo
```

va a ser siempre *True*, dado que o bien el pollo está frito o bien el pollo está crudo (en el mundo de la programación nunca hay pollos a medio cocinar). Relajando un poco las cosas, si nos da un poco igual si el pollo está frito, está crudo o las dos cosas (¿un pollo cuántico?), podemos decir que

```
pollofrito or pollocrudo
```

es siempre *True*. Pero eso sí, ¡bajo ningún concepto!, ni por encima del cadáver del pollo, podemos conseguir que

```
pollofrito and pollocrudo
```

sea cierto. Esto va a ser siempre *False*. O está frito o está crudo. Pero hemos quedado que en este mundo binario nunca vamos a tener ambas cosas a la vez.

Si sólo nos comemos un pollo cuando está frito (aunque habrá alguna tribu que tal vez se los coma crudos) y el valor *polloingerido* representa que nos hemos comido el pollo, podríamos ver que

```
pollofrito and polloingerido
```

bien podría ser cierto. Pero desde luego

```
pollocrudo and polloingerido
```

tiene mas bien aspecto de ser falso.

Los booleanos son extremadamente importantes como estamos empezando a ver. Son lo que nos permite que un programa tome decisiones en función de cómo están las cosas. Por lo tanto tenemos que dominarlos realmente bien para hacer buenos programas.

Tenemos también una gama de operadores que ya conoces de matemáticas que aquí producen como resultado un valor de verdad. Estos son los operadores de comparación, llamados así puesto que se utilizan para comparar valores numéricos entre sí:

```
<
>
<=
>=
=
/=
```

Estos corresponden a las relaciones *menor que*, *mayor que*, *menor o igual que*, *mayor o igual que*, *igual a* y *distinto a*. Así por ejemplo, `3 < 2` es *False*, pero `2 <= 2` es *True*.

Por ejemplo, sabemos que si *a* y *b* son dos *Integer*, entonces

```
(a < b) or (a = b) or (a > b)
```

va a ser siempre *True*. No hay ninguna otra posibilidad. Además,

```
(a < b) xor (a >= b)
```

va a ser también *True* en todos los casos (no puede suceder que un número sea a la vez menor que otro y mayor o igual que éste, y además una de las dos cosas es cierta siempre).

Cuando queramos que un programa tome una decisión adecuada al considerar alguna condición nos vamos a tener que plantear *todos los casos posibles*. Y vamos a tener que aprender a escribir expresiones booleanas para los casos que nos interesen.

Pero cuidado, los números reales se almacenan como aproximaciones. Piénsese por ejemplo en como si no podríamos almacenar el número π, que tiene infinitas cifras decimales. Por ello no es recomendable comparar números reales empleando ‘‘`=`’’ o ‘‘`/=`’’. En su lugar, suele ser mucho mejor ver si la diferencia, en valor absoluto, entre un número y aquel con el que lo queremos comparar es menor que un épsilon (una diferencia dada entre los dos valores). Por ejemplo,

```
abs (3.1415 - Pi) < 0.005
```

estaría bien si sólo nos interesa ver si 3.1415 es nuestro *Pi* con dos decimales de precisión. Pero posiblemente

```
(3.1415 = Pi) = False
```

La comparación de igualdad y desigualdad suele estar disponible en general para todos los tipos de datos. Las comparaciones que requieren un orden entre los elementos comparados sólo suelen estar disponibles en tipos de datos numéricos.

Quizá sorprenda que también pueden compararse caracteres. Por ejemplo, esta expresión es cierta, *True*, cuando *x* es un carácter que corresponde a una letra mayúscula:

```
('A' <= x) and (x <= 'Z')
```

Las letras mayúscula están en el juego de caracteres en posiciones consecutivas. Igual sucede con las minúsculas. Los dígitos también. Pero cuidado, esta expresión no tiene mucho sentido:

```
x <= '0'
```

No sabemos qué caracteres hay antes del carácter `'0'` en el código ASCII (salvo si miramos la tabla que describe dicho código).

Otro ejemplo más. Debido al tipado estricto de datos en Ada no podemos comparar

```
3 < '0'
```

dado que no hay **concordancia de tipos** entre `3` y `'0'`. Esto es, dado que los tipos no son compatibles para usarlos en el mismo operador ‘‘`<`’’.

2.6. Años bisiestos

Podemos combinar todos estos operadores en expresiones más complicadas. Y así lo haremos con frecuencia. Por ejemplo, sea *a* un entero que representa un año, como 2008 o cualquier otro. Esta expresión es *True* si el año *a* es bisiesto:

```
(a mod 4) = 0 and ((a mod 100) /= 0 or ((a mod 400) = 0))
```

¿De dónde hemos sacado esto? Los años múltiplos de 4 son bisiestos, excepto los que son múltiplos de 100. Salvo por que los múltiplos de 400 lo son. Si queremos una expresión que sea cierta cuando el año es múltiplo de 4 podemos fijarnos en el módulo entre 4. Este será 0, 1, 2 o 3. Para los múltiplos de 4 va a ser siempre 0. Luego

```
(a mod 4) = 0
```

es *True* si *a* es múltiplo de 4. Queremos excluir de nuestra expresión aquellos que son múltiplos de 100, puesto que no son bisiestos. Podemos escribir entonces una expresión que diga que *el módulo entre 4 es cero y no es cero el módulo entre 100*. Esto es:

```
(a mod 4) = 0 and (not ((a mod 100) = 0))
```

Esta expresión es cierta para todos los múltiplos de 4 salvo que sean también múltiplos de 100. Pero esto es igual que

```
(a mod 4) = 0 and (a mod 100) /= 0
```

que es más sencilla y se entiende mejor. Pero tenemos que hacer que para los múltiplos de 400, a pesar de ser múltiplos de 100, la expresión sea cierta. Dichos años son bisiestos y tenemos que considerarlos. Veamos: suponiendo que *a* es múltiplo de 4, lo que es un candidato a año bisiesto... Si *a* es no múltiplo de 100 o *a* es múltiplo de 400 entonces tenemos un año bisiesto. Esto lo podemos escribir como

```
(a mod 100) /= 0 or (a mod 400 = 0)
```

pero tenemos naturalmente que exigir que nuestra suposición inicial (que *a* es múltiplo de 4) sea cierta:

```
(a mod 4) = 0 and ((a mod 100) /= 0 or ((a mod 400) = 0))
```

¿Cómo hemos procedido para ver si *a* es bisiesto? Lo primero ha sido definir el problema: Queremos *True* cuando *a* es bisiesto y *False* cuando no lo es. Ese es nuestro problema.

Una vez hecho esto, hemos tenido que saber cómo lo haríamos nosotros. En este caso lo mas natural es que no sepamos hacerlo (puesto que normalmente miramos directamente el calendario y no nos preocupamos de calcular estas cosas). Así pues tenemos que aprender a hacerlo nosotros antes siquiera de pensar en programarlo. Tras buscar un poco, aprendemos mirando en algún sitio (¿Google?) que

> *‘‘Los años múltiplos de 4 son bisiestos, excepto los que son múltiplos de 100. Salvo por que los múltiplos de 400 lo son.’’*

Nuestro plan es escribir esta definición en Ada y ver cuál es su valor (cierto o falso). Ahora

tenemos que escribir el programa. Lo más importante es en realidad escribir la expresión en Ada que implementa nuestro plan. Procediendo como hemos visto antes, llegamos a nuestra expresión

```
(a mod 4) = 0 and ((a mod 100) /= 0 or ((a mod 400) = 0))
```

¡Ahora hay que probarlo! ¿Será e 2527 un año bisiesto? ¿Y el año 1942? Tomamos prestado el programa para saludar a π y lo cambiamos para que en lugar de saludar a π escriba esta expresión. Por el momento no sabemos lo necesario para hacer esto (dado que no hemos visto ningún programa que escriba valores de verdad). En cualquier caso, este sería el programa resultante.

esbisiesto.adb

```
1     --
2     -- ¿Es 2527 un año bisiesto?
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure EsBisiesto is

10          -- necesitamos esto para escribir booleanos
11          package BoolIO is new Enumeration_IO(Boolean);
12          use BoolIO;

14          -- A es el año que nos interesa.
15          A: constant Integer := 2527;

17          -- Esta constante booleana será cierta si A es
18          -- bisiesto.
19          EsBisiesto: constant Boolean :=
20               (A mod 4) = 0 and
21               ((A mod 100) /= 0 or (A mod 400 = 0));

23    begin
24          -- Esta sentencia escribe el valor booleano
25          Put(EsBisiesto);
26    end;
—
```

Lo que resta es compilar y ejecutar el programa:

```
; ada esbisiesto.adb
; esbisiesto
FALSE
```

Cuando hemos ejecutado el programa Ada ha hecho lo siguiente:

1 Ha calculado el valor de las constantes que hemos definido.

2 Ha ejecutado las sentencias en el cuerpo del programa. En este caso, estas sentencias han escrito el valor de nuestra constante como resultado (salida) del programa.

Modifica tú este programa para ver si 1942 es bisiesto o no. Intenta cambiarlo para que un sólo programa te diga si dos años que te gusten son bisiestos.

2.7. Más sobre expresiones

¿Cómo calcula Ada las constantes? En cualquier definición o en cualquier sentencia que utilice una expresión Ada **evalúa** el valor de la expresión durante la ejecución de la sentencia (o de la definición). O lo que es lo mismo, Ada calcula un valor correspondiente a la expresión. En nuestro caso Ada ha calculado un valor de tipo *Boolean* para la expresión a partir de la cual hemos definido la constante *EsBisiesto*.

Las expresiones se evalúan siempre *de dentro hacia afuera*, aunque no sabemos si se evalúan de derecha a izquierda o de izquierda a derecha. Esto quiere decir que las expresiones se evalúan haciendo caso de los paréntesis (y de la precedencia de los operadores) de tal forma que primero se hacen los cálculos más interiores o **anidados** en la expresión. Por ejemplo, si partimos de

```
(a mod 4) = 0 and ((a mod 100) /= 0 or ((a mod 400) = 0))
```

y *a* tiene como valor `444`, entonces Ada evalúa

```
(444 mod 4) = 0 and ((444 mod 100) /= 0 or ((444 mod 400) = 0))
```

Aquí, Ada va a calcular primero `444` *mod* `4` o bien `444` *mod* `100` o bien `444` *mod* `400`. Estas sub-expresiones son las más internas o más anidadas de la expresión. Supongamos que tomamos la segunda. Esto nos deja:

```
(444 mod 4) = 0 and (44 /= 0 or ((444 mod 400) = 0))
```

Ada seguiría eliminando paréntesis (evaluándolos) de dentro hacia afuera del mismo modo, calculando...

```
(444 mod 4) = 0 and (44 /= 0 or (44 = 0))
(444 mod 4) = 0 and (44 /= 0 or False)
(444 mod 4) = 0 and (True or False)
0 = 0 and (True or False)
True and (True or False)
True and True
True
```

Si no tenemos paréntesis recurrimos a la precedencia para ver en qué orden se evalúan las cosas. En la figura 2.2 mostramos todos los operadores, de mayor a menor precedencia (los que están en la misma fila tienen la misma precedencia).

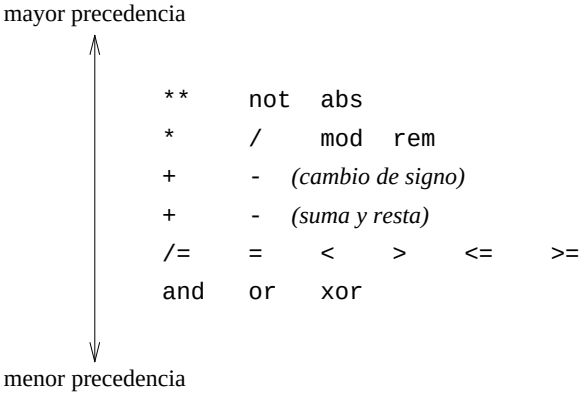


Figura 2.2: *Precedencia de operadores en Ada.*

Si tomamos por ejemplo la expresión

```
not False and (False or True)
```

podemos ver que *not* va antes que *and*; por lo que en realidad tenemos

```
((not False) and (False or True))
```

Para evaluarla nos fijamos en partes de la expresión de dentro de los paréntesis hacia afuera y vamos reescribiendo la expresión de tal forma que cambiamos cada parte por su valor. Por ejemplo:

```
((not False) and (False or True))
(True and (False or True))
(True and True)
True
```

Ada no permite mezclar distintos operadores lógicos con la misma precedencia en una expresión sin usar paréntesis para indicar el orden de su evaluación. Por ejemplo, la siguiente expresión no compila:

```
not False and False or True  ¡incorrecto!
```

En ese caso, obtendremos un error de compilación como este:

```
; ada malo.adb
malo.adb:11:26: mixed logical operators in expression
```

Por cierto, hay dos leyes muy útiles en expresiones booleanas que son las denominadas **leyes de Morgan** y que podemos utilizar para simplificar expresiones booleanas. Según estas leyes, resulta que las siguientes expresiones son equivalentes:

```
not (a and b)
(not a) or (not b)
```

Y las siguientes son también equivalentes:

```
not (a or b)
(not a) and (not b)
```

Puedes comprobarlo. Sugerimos intentar eliminar los *not* de las condiciones en los programas, puesto que los humanos entienden peor las negaciones que las afirmaciones ¿No te parece que no sucede que no es cierto? O tal vez deberíamos decir... ¿No es cierto?

2.8. Atributos

En Ada existen funciones, llamadas **atributos**, que resultan muy útiles para construir expresiones (Otros lenguajes utilizan constantes y funciones). Estas funciones pueden ayudarnos a obtener valores que dependen de un tipo de datos determinado. Se escriben indicando el nombre del tipo y después el nombre del atributo, posiblemente con algún argumento entre paréntesis. Sólo mencionaremos alguno de ellos, otros irán saliendo durante el curso. Por ejemplo,

```
Integer'First
```

corresponde al menor entero disponible en Ada. Sabemos que la sintaxis para escribir los atributos, con una ‘‘’’ entre el nombre del tipo y el nombre del atributo, es realmente espantosa. Pero no tenemos la culpa de esto. Continuando,

```
Integer'Last
```

corresponde al mayor entero disponible. Los enteros se almacenan en el ordenador con un número finito de bits y por tanto tienen límites, al contrario que los enteros matemáticos. El atributo

```
Character'Pos(x)
```

es un valor de tipo *Integer* que corresponde a la posición del carácter *x* en la lista de todos los

caracteres (esto es, a su código ASCII). Por otro lado

```
Character'Val(n)
```

es un valor de tipo *Character* que corresponde al carácter situado en la posición *n* dentro de la lista de todos los caracteres. Los atributos

```
Character'Pred(x)
Character'Succ(x)
```

tienen como valor los caracteres predecesor (anterior) y sucesor (siguiente) al carácter *x* en el código. Por ejemplo,

```
Character'Pred(x)
```

tendrá como valor `'A'` siempre que *x* sea `'B'`.

2.9. Longitud de una circunferencia

Vamos a poner todo esto en práctica. Queremos un programa que escriba la longitud de una circunferencia de radio *r*. El problema es, dado *r*, calcular $2\pi r$, naturalmente. Podríamos modificar el último programa que hemos hecho para conseguir esto. Pero vamos a escribirlo desde el principio. Lo primero que necesitamos es escribir la estructura de nuestro programa. Podemos empezar por...

```
1  --
2  -- Programa para calcular la longitud de
3  -- la circunferencia en un círculo de radio r
4  --

6  with Ada.Text_IO;
7  use Ada.Text_IO;

9  procedure CalculoCircunferencia is
10 begin
11     null;
12 end;
```

No es que haga mucho. De hecho, hemos utilizado la sentencia nula, *null*, como cuerpo del programa. Esto quiere decir que Ada no va a ejecutar nada cuando ejecute el cuerpo del programa. Ahora compilamos y ejecutamos el programa para ver al menos que está bien escrito.

Lo siguiente que necesitamos es nuestra constante π. Podemos utilizar la que teníamos escrita en otros programas y ponerla al principio en la zona de declaraciones de constantes.

```
1  --
2  -- Programa para calcular la longitud de
3  -- la circunferencia en un círculo de radio r
4  --

6  with Ada.Text_IO;
7  use Ada.Text_IO;

9  procedure CalculoCircunferencia is

11     -- Número π
12     Pi: constant Float := 3.1415926;
```

```
14 begin
15     null;
16 end;
```

De nuevo compilamos y ejecutamos el programa. Al menos sabremos si tiene errores sintácticos o no. Si los tiene será mucho más fácil encontrar los errores ahora que luego.

Ahora podríamos escribir otra constante para nuestro radio, *r*, y otra más para la expresión que queremos calcular. Podríamos incluir esto justo debajo de la declaración para *Pi*:

```
1  r: constant Float := 3;
2  LongitudCircunferencia: constant Float := 2.0 * Pi * r;
```

Por último necesitamos escribir el resultado de nuestro cálculo. Para ello cambiamos *null* por

```
Put(LongitudCircunferencia)
```

en el cuerpo del programa. Pero falta algo... Tenemos que escribir esas dos lineas que particularizaban una librería para escribir reales. El programa resultante es como sigue:

```
1  --
2  -- Programa para calcular la longitud de
3  -- la circunferencia en un círculo de radio r
4  --

6  with Ada.Text_IO;
7  use Ada.Text_IO;

9  procedure CalculoCircunferencia is

11     -- Número π
12     Pi: constant Float := 3.1415926;
13     -- radio que nos interesa
14     r: constant Float := 3;
15     LongitudCircunferencia: constant Float := 2.0 * Pi * r;

17     -- Necesario para escribir Floats.
18     package Float_InOut is new Float_IO(Float);
19     use Float_InOut;

21 begin
22     Put(LongitudCircunferencia);
23 end;
```

Si lo compilamos de nuevo y lo ejecutamos tenemos el resultado.

```
; ada calculocircunferencia.adb
; calculocircunferencia
  1.25664E+01
```

Este programa está bien. Pero lo suyo es que ya que la longitud de una circunferencia es en realidad una función, el programa utilice una función para calcularla (como el programa que utilizamos para ver la anatomía de un programa al principio del curso). Podemos copiar la función que aparecía en ese programa y adaptarla a nuestros propósitos. Ésta nos quedaría así:

```
1  -- calcular longitud de circunferencia para
2  -- circulo de radio r.
3  function LogitudCircunferencia(r: Float) return Float is
4  begin
5      return 2.0 * Pi * r;
6  end;
```

La sintaxis puede ser rara, pero es fácil de entender. Esto define una función llamada *LongitudCircunferencia* a la que podemos dar un número de tipo *Float* para que nos devuelva otro número de tipo *Float*. El número que le damos a la función lo llamamos *r*. Entre *begin* y *end* tenemos que escribir cómo se calcula el valor de la función. La sentencia *return* hace que la función devuelva el valor escrito a su derecha ($2\pi r$) cada vez que le demos un número *r*. Si utilizamos esto, el programa quedaría como sigue:

calculocircunf.adb

```
1     --
2     -- Programa para calcular la longitud de
3     -- la circunferencia en un círculo de radio r
4     --

6     with Ada.Text_IO;
7     use Ada.Text_IO;

9     procedure CalculoCircunf is

11          -- Número π
12          Pi: constant Float := 3.1415926;

14          -- Necesario para escribir Floats.
15          package Float_InOut is new Float_IO(Float);
16          use Float_InOut;

18          -- calcular longitud de circunferencia para
19          -- circulo de radio r.
20          function LogitudCircunferencia(r: Float) return Float is
21          begin
22               return 2.0 * Pi * r;
23          end;

25    --
26    -- Constantes de pruebas.
27    --
28          Prueba1: constant Float := LogitudCircunferencia(2.0);

30    begin
31          -- Ejecución de pruebas.
32          Put("Longitud para r=2.0: ");
33          Put(Prueba1);
34          New_Line;
35    end;
```

Hemos incluido como constante para nuestras pruebas la constante *Prueba1* que tomará como valor la longitud de la circunferencia de radio 2.0. En el cuerpo del programa hacemos que el programa escriba un mensaje explicativo, el valor de nuestro cálculo y pase a la línea siguiente.

Esta es la compilación y ejecución del programa:

```
; ada calculocircunferencia
; calculocircunferencia
Longitud para r=2.0:  1.25664E+01
```

En adelante utilizaremos funciones para realizar nuestros cálculos.

Problemas

1 Modifica el último programa que hemos mostrado para que calcule el área de un circulo de radio dado en lugar de la longitud de la circunferencia dado el radio. *Presta atención a las normas de estilo.* Esto quiere decir que tendrás que cambiar el nombre del programa y tal vez otras cosas además de simplemente cambiar el cálculo.

2 Escribe en Ada las siguientes expresiones. Recuerda que para que todos los números utilizados sean números reales has de poner siempre su parte decimal, aunque ésta sea cero. En algunos casos te hará falta utilizar funciones numéricas elementales en Ada, tales como la raíz cuadrada. Prueba a buscar en el libro de texto cómo se utilizan. Prueba también a buscar en Google ‘‘raíz cuadrada en ada’’, por ejemplo.

a)

$$2.7^2 + -3.2^2$$

b) Para al menos 3 valores de *r*,

$$\frac{4}{3}\pi r^3$$

c) El factorial de 5, que suele escribirse como 5!. El factorial de un número natural es dicho número multiplicado por todos los naturales menores que el hasta el 1. (Por definición, 0! se supone que tiene como valor 1).

d) Dados $a = 4$ y $b = 3$

$$\binom{a}{b} = \frac{a!}{b!(a-b)!}$$

e)

$$\sqrt{\pi}$$

f) Para varios valores de x entre 0 y 2π,

$$sen^2 x + \cos^2 x$$

g) Siendo $x = 2$.

$$\frac{1}{\sqrt{3.5x^2 + 4.7x + 9.3}}$$

3 Por cada una de las expresiones anteriores escribe un programa en Ada que imprima su valor.

a) Hazlo primero declarando una constante de prueba que tenga como valor el de la expresión a calcular.

b) Hazlo utilizando una función para calcular la expresión. En los casos en que necesites funciones de más de un argumento puedes utilizar el ejemplo que sigue:

```
1  function Sumar(a: Float; b:Float) return Float is
2  begin
3      return a + b;
4  end;
```

4 Para cada una de las expresiones anteriores indica cómo se evalúan éstas, escribiendo cómo queda la expresión tras cada paso elemental de evaluación hasta la obtención de un único valor como resultado final.

3 — Resolución de problemas

3.1. Problemas y funciones

El propósito de un programa es resolver un problema. Y ya hemos resuelto algunos. En el último capítulo utilizamos una función para calcular la longitud de una circunferencia, por ejemplo. Ahora vamos a prestar más atención a lo que hicimos para ver cómo resolver problemas nuevos y problemas más difíciles.

Lo primero que necesitábamos al realizar un programa era definir el problema. Pensando en esto, prestemos atención ahora a esta línea de código que ya vimos antes:

```
function LongitudCircunferencia(r: Float) return Float
```

Esta linea forma parte de la definición de la función Ada *AreaCirculo*; se la denomina **cabecera de función** para la función *AreaCirculo*. El propósito de esta línea es indicar:

1 Cómo se llama la función.

2 Qué necesita la función para hacer sus cálculos.

3 Qué devuelve la función como resultado (que tipo de valor tiene la función).

Si lo piensas, ¡La definición de un problema es justo esto!

1 Cuál es el problema que resolvemos.

2 Qué necesitamos para resolverlo.

3 Qué tendremos una vez esté resuelto.

Luego

la definición de un problema es la cabecera de una función

En general, decimos que la definición de un problema es la cabecera de un subprograma (dado que hay también otro tipo de subprogramas, llamados procedimientos, como veremos más adelante). Esto es muy importante debido a que lo vamos a utilizar a todas horas mientras programamos.

Por ejemplo, si queremos calcular el área de un círculo nuestro problema es, dado un radio *r*, calcular el valor de su área (πr^2). Por lo tanto, el problema consiste en tomar un número real, *r*, y calcular otro número real.

El nombre del problema en Ada podría ser *AreaCirculo*. Si queremos programar este problema ya sabemos que al menos tenemos que definir una función con este nombre:

```
function AreaCirculo...
```

Puede verse que el identificador que da nombre a la función se escribe tras la palabra reservada *function*. Dicho identificador debe ser un nombre descriptivo del resultado de la función (¡Un nombre descriptivo del problema!).

Ahora necesitamos ver qué necesitamos para hacer el trabajo. En este caso el radio *r*, que es un número real. En Ada podríamos utilizar el identificador *r* (dado que en este caso basta para saber de qué hablamos). Además, sabemos ya que este valor tiene que ser de tipo *Float*.

La forma de suministrarle valores a la función para que pueda hacer su trabajo es definir sus **parámetros**. En este caso, un único parámetro *r* de tipo *Float*. Los parámetros hay que declararlos o definirlos entre paréntesis, tras el nombre de la función (Piensa que en matemáticas usarías

$f(x)$ para una función de un parámetro).

```
function AreaCirculo(r: Float)...
```

Por último necesitamos definir de qué tipo va ser el valor resultante cada vez que utilicemos la función. En este caso la función tiene como valor un número real. Decimos que devuelve un número real:

```
function AreaCirculo(r: Float) return Float
```

¡Esta línea es la definición en Ada de nuestro problema! Hemos dado el primer paso de los que teníamos que dar para resolverlo. En cuanto lo hayamos hecho podremos escribir expresiones como *AreaCirculo(3.2)* para calcular el área del circulo con radio 3.2. Al hacerlo, suministramos 3.2 como valor para *r* en la función (véase la figura 3.1); El resultado de la función es otro valor correspondiente en este caso a πr^2.

Deberás recordar que que se llama **argumento** al valor concreto que se suministra a la función (por ejemplo, 3.2) cuando se produce una llamada (cuando se evalúa la función) y **parámetro** al identificador que nos inventamos para nombrar el argumento que se suministrará cuando se produzca una llamada (por ejemplo *r*). En la figura 3.1 los argumentos están representados por valores sobre las flechas que se dirigen a los parámetros (los cuadrados pequeños a la entrada de cada función).

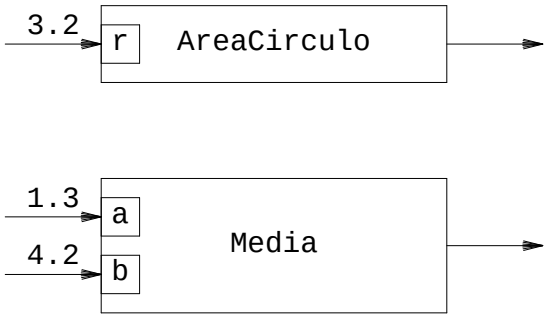


Figura 3.1: *Varias funciones Ada. Tienen argumentos y devuelven resultados.*

En muchas ocasiones un problema requerirá de varios elementos para su resolución. Por ejemplo, obtener la media de dos números requiere dos números. Si queremos la media de 1.3 y 4.2 podríamos escribir `Media(1.3,4.2)`. En tal caso se procede del mismo modo para definir el problema (la cabecera de la función), separando los argumentos con ‘‘`;`’’. Por ejemplo:

```
function Media(num1: Float; num2: Float) return Float
```

Volviendo a nuestro problema de ejemplo, resta implementar la solución. La forma de hacer esto en Ada es escribir el **cuerpo** de la función. Para hacerlo supondremos que tenemos ya definida la constante *Pi*. Si no es así, como ahora nos vendría bien tenerla, nos la inventamos justo ahora y no hay mayor problema. La función con su cabecera y cuerpo queda como sigue:

```
1  function AreaCirculo(r: Float) return Float is
2  begin
3      return Pi * r ** 2;
4  begin;
```

La palabra reservada *is* indica que sigue la implementación de la función. La sentencia *return*, como sabemos, hace que la función devuelva un valor como resultado. Como la definición entera de la función se considera una declaración (igual que cuando declaramos constantes), tenemos que terminarla con un ‘‘`;`’’.

Todo esto es similar a lo que hemos hecho hasta el momento para escribir programas. No obstante, ahora estamos implementando un subprograma y no el programa principal. Pero por lo demás es igual.

Ahora podríamos hacer un programa completo para calcular el área de un círculo, o de varios círculos. Este podría ser tal programa.

calculoaarea.adb

```
1     --
2     -- Programa para calcular el area
3     -- de un circulo de radio r
4     --

6     with Ada.Text_IO;
7     use Ada.Text_IO;

9     procedure CalculoAarea is

11          -- Numero π
12          Pi: constant Float := 3.1415926;

14          -- Necesario para escribir Floats.
15          package Float_InOut is new Float_IO(Float);
16          use Float_InOut;

18          function AreaCirculo(r: Float) return Float is
19          begin
20               return Pi * r ** 2;
21          end;

23    --
24    -- Constantes de pruebas.
25    --
26          Radio1: constant Float := 3.2;
27          Radio2: constant Float := 4.0;

29          Prueba1: constant Float := AreaCirculo(Radio1);
30          Prueba2: constant Float := AreaCirculo(Radio2);

32    begin
33          -- Ejecucion de pruebas.
34          Put(Prueba1);
35          New_Line;
36          Put(Prueba2);
37          New_Line;
38    end;
```

Dado que la función es obvio lo que hace, lo que necesita y lo que devuelve a la luz de los nombres que hemos utilizado, nos parece innecesario escribir un comentario aclarando lo que hace la función.

Al ejecutar este programa, Ada comenzará por evaluar cada una de las expresiones que hemos utilizado para definir las constantes en las líneas 12, 26, 27, 29 y 30. Cuando llegue el momento de evaluar *AreaCirculo(Radio1)* Ada ‘‘llamará’’ a la función *AreaCirculo* utilizando *Radio1* (esto es, 3.2) como argumento (como valor) para el parámetro *r*. La función tiene como cuerpo una única sentencia que hace que el valor resultante de la función sea πr^2.

Una vez inicializadas y definidas las constantes, Ada ejecutará las sentencias del cuerpo del programa principal que se limitan a escribir los valores de nuestras pruebas una por línea:

```
; calculoarea
 3.21699E+01
 5.02655E+01
```

Podemos utilizar llamadas a función en cualquier lugar donde podemos utilizar un valor del tipo que devuelve la función considerada. Por ejemplo, *AreaCirculo(x)* puede utilizarse en cualquier sitio en que podamos emplear un valor de tipo *Float*. Sólo podemos utilizar la función a partir del punto en el programa en que la hemos definido. Nunca antes. Lo mismo sucede con cualquier otra declaración.

Veamos un ejemplo. Supongamos que en alguna sentencia del programa principal utilizamos la expresión

```
(3.0 + AreaCirculo(2.0 * 5.0 - 1.0)) / 2.0
```

Cuando Ada encuentre dicha expresión la evaluará, procediendo como hemos visto para otras expresiones:

```
(3.0 + AreaCirculo(2.0 * 5.0 - 1.0)) / 2.0
(3.0 + AreaCirculo(10.0 - 1.0)) / 2.0
(3.0 + AreaCirculo(9.0)) / 2.0
```

En este punto Ada deja lo que está haciendo y llama a *AreaCirculo* utilizando 9.0 como valor para el parámetro *r*. Eso hace que Ada calcule

```
Pi * 9.0 ** 2
```

lo que tiene como valor `2.54469E+02`. La sentencia *return* de la función hace que se devuelva este valor. Ahora Ada continúa con lo que estaba haciendo, evaluando nuestra expresión:

```
(3.0 + 2.54469E+02) / 2.0
2.57469E+02 / 2.0
1.28734E+02
```

Resumiendo todo esto, podemos decir que una función en Ada es similar a una función matemática. A la función matemática se le suministran valores de un dominio origen y la función devuelve como resultado valores en otro dominio imagen. En el caso de Ada una función recibe uno o más valores y devuelve un único valor de resultado. Por ejemplo, si *AreaCirculo* es una función que calcula el área de un circulo dado un radio, entonces *AreaCirculo(3.0)* es un valor que corresponde al área del círculo de radio 3. Cuando Ada encuentra *AreaCirculo(3.0)* en una expresión Ada procede a evaluar la función llamada *AreaCirculo* y eso deja como resultado un valor devuelto por la función.

Desde este momento sabemos que

un subproblema lo resuelve un subprograma

y definiremos un subprograma para cada subproblema que queramos resolver. Hacerlo así facilita la programación, como ya veremos. Por supuesto necesitaremos un programa principal que tome la iniciativa y haga algo con el subprograma (llamarlo), pero nos vamos a centrar en los subprogramas la mayor parte del tiempo.

En muchos problemas vamos a tener parámetros que están especificados por el enunciado (como ‘‘100’’ en ‘‘calcular los 100 primeros números primos’’). En tal caso lo mejor es definir constantes para los parámetros del programa. La idea es que

para cambiar un dato debe bastar cambiar una constante

y volver a compilar y ejecutar el enunciado. Esto no sólo es útil para modificar el programa, también elimina errores.

3.2. Declaraciones

Hemos estado utilizando declaraciones todo el tiempo sin saber muy bien lo que es esto y sin prestar mayor atención a este hecho. Es hora de verlo algo más despacio.

Normalmente se suele distinguir entre **declarar** algo en un programa y **definir** ese algo. Por ejemplo, la cabecera de una función *declara* que la función existe, tiene unos parámetros y un resultado. El cuerpo de una función *define* cómo se calcula la función.

En el caso de las constantes que hemos estado utilizando durante el curso, las líneas del programa que las *declaran* están también definiéndolas (dado que definen que valor toman).

Un objeto (una contante, un subprograma, un tipo, etc. sólo puede utilizarse desde el punto en que se ha declarado hasta el final del programa (o subprograma) en que se ha declarado. Fuera de este **ámbito** o zona del programa el objeto no es **visible** y es como si no existiera. Volveremos sobre esto más adelante.

Habitualmente resulta útil declarar constantes cuando sea preciso emplear constantes bien conocidas (o literales que de otro modo parecerían números mágicos o sin explicación alguna). Para declarar una constante se procede como hemos visto durante el curso:

```
Pi: constant Float := 3.1415926;
```

Primero se escribe el identificador de la constante, *Pi*, seguido de ‘‘`:`’’ y el nombre del tipo de datos al que pertenece la constante. En este caso *Float*. Por último, y tras ‘‘`:=`’’, se escribe la expresión que define el valor de la constante; como de costumbre, también hay que escribir un ‘‘`;`’’ para terminar la declaración. Una vez hemos declarado la constante *Pi* podemos utilizar su identificador en expresiones en cualquier lugar del programa desde el punto en que se ha declarado la constante hasta el final del programa.

3.3. Problemas de solución directa

Hay muchos problemas que ya podemos resolver mediante un programa. Concretamente, todos los llamados *problemas con solución directa*. Como su nombre indica son problemas en que no hay que decidir nada, tan sólo efectuar un cálculo según indique algún algoritmo o teorema ya conocido.

Por ejemplo, la siguiente función puede utilizarse como (sub)programa para resolver el problema de calcular el factorial de 5.

```
1  function FactorialDe5 return Integer is
2  begin
3      return 5 * 4 * 3 * 2 * 1;
4  end;
```

Ha bastado seguir estos pasos:

1 Buscar la definición de factorial de un número.

2 Definir el problema en Ada (escribiendo la cabecera de la función).

3 Escribir la expresión en Ada que realiza el cálculo, en la sentencia *return*.

4 Compilarlo y probarlo.

5 Depurar los errores cometidos.

Procederemos siempre del mismo modo.

Para problemas más complicados hay que emplear el optimismo. La clave de todo radica en suponer que tenemos disponible todo lo que nos pueda hacer falta para resolver el problema que nos ocupa (salvo la solución del problema que nos ocupa, claro está). Una vez resuelto el problema, caso de que lo que nos ha hecho falta para resolverlo no exista en realidad, tenemos que proceder a programarlo. Para ello volvemos a aplicar la misma idea.

Esto se entenderá fácilmente si intentamos resolver un problema algo más complicado. Supongamos que queremos calcular el valor del volumen de un sólido que es un cilindro al que se ha perforado un hueco cilíndrico, tal y como muestra la figura 3.2.

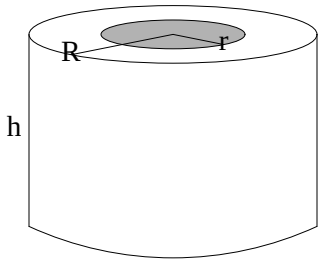


Figura 3.2: *Un sólido cilíndrico perforado por un cilindro.*

Lo primero es definir el problema. ¿Qué necesitamos saber para que esté definido nuestro sólido? Necesitamos la altura del prisma, *h* en la figura. También el radio del cilindro que hemos taladrado, *r* en la figura, y el radio del cilindro, *R* en la figura. ¿Qué debemos producir como resultado? El valor del volumen. Luego...

```
-- Calcula el volumen de un cilindro de altura y radio (rmax)
-- conocidos taladrado por otro cilindro de radio dado (rmin)
function VolCilHueco(altura: Float; rmin: Float; rmax: Float) return Float
```

es la definición de nuestro problema. Donde hemos utilizado *altura* para *h*, *rmin* para *r*, y *rmax* para *R*.

¡Y ahora somos optimistas! Si suponemos que ya tenemos calculado el área de la base, basta multiplicarla por la altura para tener nuestro resultado. En tal caso, lo suponemos y hacemos nuestro programa.

```
1  -- Calcula el volumen de un cilindro de altura y radio (rmax)
2  -- conocidos taladrado por otro cilindro de radio dado (rmin)
3  function VolCilHueco(altura: Float;
4                       rmin: Float;
5                       rmax: Float) return Float is
6  begin
7      return ¿¿AreaBase?? * altura;
8  end;
```

Verás que hemos escrito la cabecera de la función en más de una línea, con un parámetro por línea. Esto se hace siempre que la lista de parámetros es larga como para caber cómodamente en una única línea. Presta atención también a la tabulación de los argumentos.

El área de la base es en realidad el área de una corona circular, de radios *rmin* y *rmax* (o *r* y *R*). Luego el problema de calcular el área de la base es el problema de calcular el área de una corona circular (como la que muestra la figura 3.3). Este problema lo definimos en Ada como la cabecera de función:

```
-- Area de corona circular dados los radios interno y externo
function AreaCorona(rmin: Float; rmax: Float) return Float;
```

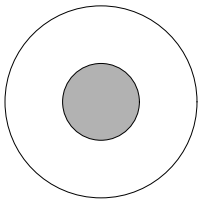


Figura 3.3: *Una corona circular (un círculo con un agujero circular en su centro).*

Por lo que en realidad nuestro programa debe ser:

```
1  -- Calcula el volumen de un cilindro de altura y radio (rmax)
2  -- conocidos taladrado por otro cilindro de radio dado (rmin)
3  function VolCilHueco(altura: Float;
4                       rmin: Float;
5                       rmax: Float) return Float is
6  begin
7      return AreaCorona(rmin, rmax) * altura;
8  end;
```

Y ya tenemos programado nuestro problema. Bueno, en realidad ahora tendremos que programar *AreaCorona*. Siendo optimistas de nuevo, si suponemos que tenemos programado el cálculo del área de un círculo, podemos calcular el área de una corona circular como la diferencia entre el área del circulo externo y el área del circulo taladrado en su centro. Luego...

```
1  -- Area de corona circular dados los radios interno y externo
2  function AreaCorona(rmin: Float; rmax: Float) return Float is
3  begin
4  return AreaCirculo(rmax) - AreaCirculo(rmin);
5  end;
```

Y para el área del círculo ya teníamos un subprograma que sabía calcularla. Nuestro programa resultante queda como sigue.

volcilindro.adb

```
1     --
2     -- Programa para calcular el volumen de un cilindro
3     -- taladrado por otro cilindro.
4     --

6     with Ada.Text_IO;
7     use Ada.Text_IO;

9     procedure VolCilindro is

11          -- Número π
12          Pi: constant Float := 3.1415926;

14          -- Necesario para escribir Floats.
15          package Float_InOut is new Float_IO(Float);
16          use Float_InOut;
```

```
18          function AreaCirculo(radio: Float) return Float is
19          begin
20               return Pi * radio ** 2;
21          end;

23          -- Area de corona circular dados los radios interno y externo
24          function AreaCorona(rmin: Float; rmax: Float) return Float is
25          begin
26               return AreaCirculo(rmax) - AreaCirculo(rmin);
27          end;

29          -- Calcula el volumen de un cilindro de altura y radio (rmax)
30          -- conocidos taladrado por otro cilindro de radio dado (rmin)
31          function VolCilHueco(altura: Float;
32                               rmin: Float;
33                               rmax: Float) return Float is
34          begin
35               return AreaCorona(rmin, rmax) * altura;
36          end;
37     --
38     -- Constantes de pruebas.
39     --
40          Prueba1: constant Float := VolCilHueco(3.0, 2.0, 3.0);

42     begin
43          -- Ejecución de pruebas.
44          Put(Prueba1);
45          New_Line;
46     end;
—
```

Un último apunte respecto a los parámetros. Aunque los mismos nombres *rmin* y *rmax* se han utilizado para los parámetros en varias funciones distintas esto no tiene por qué ser así. Dicho de otra forma: el parámetro *rmin* de *AreaCorona* no tiene nada que ver con el parámetro *rmin* de *VolCilHueco*. ¡Aunque tengan los mismos nombres!

Esto no debería ser un problema. Piensa que en la vida real personas distintas identifican objetos distintos cuando hablan de ‘‘el coche’’, aunque todos ellos utilicen la misma palabra (‘‘coche’’). En nuestro caso sucede lo mismo: cada función puede utilizar los nombres que quiera para sus parámetros.

3.4. Subproblemas

Lo que acabamos de hacer para realizar este programa se conoce como **refinamiento progresivo**. Consiste en solucionar el problema poco a poco, suponiendo cada vez que tenemos disponible todo lo que podamos imaginar (salvo lo que estamos programando, claro). Una vez programado nuestro problema nos centramos en programar los subproblemas que hemos supuesto que teníamos resueltos. Y así sucesivamente. Habrás visto que

un subproblema se resuelve con un subprograma

Sí, ya lo habíamos enfatizado antes. Pero esto es muy importante. En este caso hemos resuelto el problema comenzando por el problema en si mismo y descendiendo a subproblemas cada vez más pequeños. A esto se lo conoce como desarrollo **top-down** (de arriba hacia abajo). Es una forma habitual de programar.

En la práctica, ésta se combina con pensar justo de la forma contraria. Esto es, si vamos a calcular volúmenes y áreas y hay figuras y sólidos circulares, seguro que podemos imaginarnos ciertos subproblemas elementales que vamos a tener que resolver. En nuestro ejemplo, a lo mejor podríamos haber empezado por resolver el problema *AreaCirculo*. Podríamos haber seguido después construyendo programas más complejos, que resuelven problemas más complejos, a base de utilizar los que ya teníamos. A esto se lo conoce como desarrollo **bottom-up** (de abajo hacia arriba).

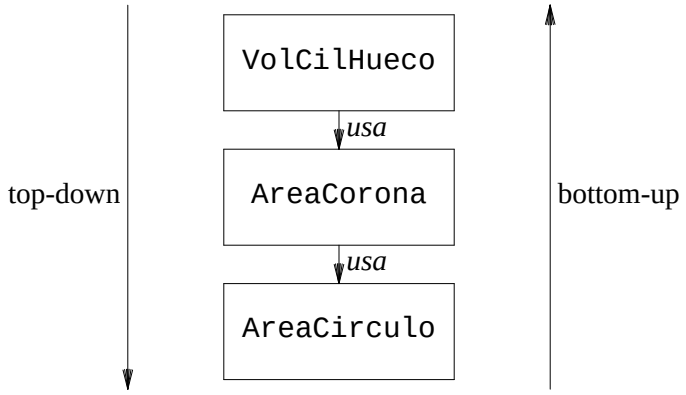


Figura 3.4: *Podemos programar de arriba-a-abajo o de abajo-a-arriba. Hay que usar ambas.*

Para programar hay que combinar las dos técnicas. Hay que organizar el programa pensando *top-down* aunque seguramente lo programemos *bottom-up*. Dicho de otro modo, abordaremos los problemas dividiéndolos en subproblemas más sencillos, pero normalmente sin programar en absoluto, hasta que tengamos un subproblema tan simple que sea fácil de programar. En ese momento lo programamos y, usándolo, continuamos programando subproblemas cada vez más complejos hasta resolver el problema original. Esto lo iremos viendo a lo largo de todo el curso.

3.5. Algunos ejemplos

Vamos a ver a continuación algunos ejemplos. En algunos casos sólo incluiremos una función que calcule el problema, y no el programa por completo. Ya sabemos cómo utilizar funciones en programas Ada y no debería haber problema en conseguir ejecutar éstos programas.

3.5.1. Escribir dígitos con espacios

Queremos un programa que escriba un número de tres dígitos con blancos entre los dígitos.

Lo primero que tenemos que hacer es ver si existe algún subprograma que, de estar disponible, nos deje hacer el trabajo de forma trivial. En este caso, de tener una función que nos suministre el tercer dígito, otra que nos suministre el segundo y otra que nos suministre el primero, bastaría usar estas tres y luego utilizar *Put* para escribir cada uno de los dígitos.

Pues bien, supondremos que tenemos estas funciones. De hecho, (astutamente, pues sabemos que tendremos que programarlas) vamos a suponer que tenemos *una* función que es capaz de darnos el dígito n-ésimo de un número. A esta función la vamos a llamar *ValorDigito*. De este modo, en lugar de tener que programar tres funciones distintas vamos a tener que programar una única función.

Considerando esta función como un subproblema, vemos que tenemos que suministrarle tanto el número como la posición del dígito que nos interesa. Una vez la tengamos podremos escribir nuestro programa.

Vamos a hacer justo eso para empezar a resolver el problema: vamos a escribir nuestro programa utilizando dicha función, aunque por el momento la vamos a programar para que siempre devuelva cero como resultado (lo que es fácil).

digitosseparados.adb

```
1     --
2     -- Escribir numero de tres digitos espaciado
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure DigitosSeparados is

10          -- Necesario para escribir Integers.
11          package Integer_InOut is new Integer_IO(Integer);
12          use Integer_InOut;

14          -- Devuelve el valor del digito en posicion dada.
15          -- La primera posicion es 1.
16          function ValorDigito(numero: Integer; pos: Integer) return Integer is
17          begin
18               -- Tendremos que programarla luego
19               return 0;
20          end;

22          -- Constantes de pruebas
23          Numero: constant Integer := 325;

25    begin
26          -- Put ya escribe espacios, no tenemos que
27          -- escribirlos nosotros.
28          Put(ValorDigito(Numero, 3));
29          Put(ValorDigito(Numero, 2));
30          Put(ValorDigito(Numero, 1));
31          New_Line;
32    end;
—
```

Ahora lo compilamos y lo ejecutamos. Sólo para ver que todo va bien por el momento.

```
; ada valordigito.adb
; valordigito
       0      0      0
```

Todo bien.

Pasemos a implementar la parte que nos falta. Ahora queremos el n-ésimo dígito de la parte entera de un número. Por ejemplo, dado 134, si deseamos el segundo dígito entonces queremos obtener 3 como resultado.

Si el problema parece complicado (aunque no lo es), lo primero que hay que hacer es **simplificarlo**. Por ejemplo, supongamos que siempre vamos a querer el segundo dígito.

Si sigue siendo complicado, lo volvemos a simplificar. Por ejemplo, supondremos que sólo queremos el primer dígito. Este parece fácil. Sabemos que cada dígito corresponde al factor de una potencia de 10 en el valor total del número. Esto es,

$$143 = 1 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0$$

Por lo tanto, el módulo entre 10 es en realidad el valor del primer dígito. Una vez que hemos visto

esto, podemos complicar el problema un poco más para resolver algo más cercano al problema original.

Consideremos el problema que habíamos considerado antes consistente en obtener el segundo dígito. Si dividimos entre 10 obtendremos el efecto de desplazar los dígitos a la derecha una unidad. Tomando a continuación el módulo entre 10 obtendremos el primer dígito, que era antes el segundo. Si queremos el tercer dígito habríamos tenido que dividir por 100 y tomar luego el módulo entre 10.

Dado que esto parece fácil ahora, podemos complicarlo un poco más y obtener una solución para nuestro problema original. Si dividimos entre 10^{n-1} y tomamos el módulo entre 10 tenemos el valor del dígito que nos interesa. En Ada podemos programar esto como una expresión de forma directa. Así pues, basta cambiar la función *ValorDigito* para que sea como sigue:

```
1  function ValorDigito(numero: Integer; pos: Integer) return Integer is
2  begin
3      return (numero / 10 ** (pos - 1)) mod 10;
4  end;
```

Y de este modo tendremos el programa completo. Si ahora lo compilamos y lo ejecutamos veremos que, en lugar de escribir ceros, el programa escribe dígitos.

```
; ada valordigito.adb
; valordigito
      3     2     5
```

¿Tenemos el programa hecho? Bueno, en principio sí. Pero merece la pena pensar un poco más y ver si podemos hacer algo mejor.

Lo único que no gusta mucho en este código es que aparezcan literales ‘‘`10`’’ de vez en cuando. Estos **números mágicos** que aparecen en los programas los hacen misteriosos. Tal vez no mucho en este caso, pero en general es así.

El ‘‘`10`’’ que aparece varias veces en el código es la base del sistema de numeración. Una opción sería cambiar la función por esta otra, declarando una constante para la base que utilizamos:

```
1  function ValorDigito(numero: Integer; pos: Integer) return Integer is
2  begin
3      return (numero / Base ** (pos - 1)) mod Base;
4  end;
```

No obstante, puede que un mismo programa necesite utilizar una función como esta pero empleando bases de numeración distintas cada vez. Así pues, lo mejor parece ser suministrarle otro argumento a la función indicando la base que deseamos emplear. Así llegamos a esta otra:

```
1  -- Devuelve el valor del digito en base y posicion dada.
2  -- La primera posicion es 1.
3  function ValorDigito(numero: Integer;
4                       pos:Integer;
5                       base:Integer) return Integer is
6  begin
7      return (numero / base ** (pos - 1)) mod base;
8  end;
```

Ahora no sólo tenemos un subprograma que puede darnos el valor de un dígito para números en base 10. También podemos obtener los valores para dígitos en cualquier otra base.

Esto lo hemos conseguido **generalizando** el subprograma que teníamos. Vimos que teníamos uno que funcionaba en el caso particular de base 10. Utilizando un parámetro en lugar del literal 10 hemos conseguido algo mucho más útil. De hecho, hicimos lo mismo cuando nos dimos cuenta de que una única función bastaba y no era preciso utilizar una distinta para cada dígito.

cuando cueste el mismo esfuerzo haremos programas más generales

3.5.2. Valor numérico de un carácter

Queremos el dígito correspondiente a un carácter numérico. Por ejemplo, como parte de un programa que lee caracteres y devuelve números enteros.

Sabemos que los caracteres numéricos están en posiciones consecutivas en el código de caracteres (del `'0'` al `'9'`). Luego si obtenemos la posición del carácter y restamos la posición del `'0'` entonces tendremos el valor numérico deseado.

```
1  function ValorCarNumerico(car: Character) return Integer is
2  begin
3      return Character'Pos(car) - Character'Pos('0');
4  end;
```

3.5.3. Carácter para un valor numérico

Esta es similar. Basta sumar el valor a la posición del carácter `'0'` y obtener el carácter en dicha posición.

```
1  function CarDigito(digito: Integer) return Character is
2  begin
3      return Character'Val(Character'Pos('0') + digito);
4  end;
```

3.5.4. Ver si un carácter es un blanco.

Un blanco es un espacio en blanco, un tabulador o un fin de línea. En Ada tenemos el paquete *ASCII* que contiene, además de otras muchas, la constante *ASCII.HT* para el tabulador (horizontal) y la constante *ASCII.LF* para el avance de línea (por ‘‘line feed’’). Como el problema consiste en ver si algo se cumple o no debería devolver un valor de verdad.

```
1  function EsBlanco(car: Character) return Boolean is
2  begin
3      return car = ' ' or car = ASCII.HT or car = ASCII.LF;
4  end;
```

3.5.5. Número anterior a uno dado módulo n

En ocasiones queremos contar de forma circular, de tal forma que tras el valor n no obtenemos $n+1$, sino 0. Decimos que contamos módulo-n, en tal caso. Para obtener el número siguiente podríamos sumar 1 y luego utilizar `mod`. Para obtener el anterior lo que haremos será sumar n

(que no cambia el valor del número, módulo-n) y luego restar 1. Así obtenemos siempre un número entre 0 y $n-1$.

```
1  function AnterioModN(num: Integer; n: Integer) return Integer is
2  begin
3      return (num + n - 1) mod n;
4  end;
```

3.6. Pistas extra

Antes de pasar a los problemas hay algunas cosas que te serán útiles.

Recuerda que la forma de proceder es definir una función para efectuar el cálculo requerido. Para efectuar las pruebas de la función puede ser preciso definir constantes auxiliares para las pruebas.

Es muy importante seguir los pasos de resolución de problemas indicados anteriormente. Por ejemplo, si no sabemos cómo resolver nosotros el problema que queremos programar, difícilmente podremos programarlo. Primero hay que definir el problema, luego podemos pensar un algoritmo, luego programarlo y por último probarlo.

Cuando sea preciso escribir valores de tipo *Integer* será preciso instanciar un paquete para entrada/salida de enteros:

```
package Integer_InOut is new Integer_IO(Integer);
use Integer_InOut;
```

Para escribir valores de tipo *Float* será preciso instanciar un paquete para entrada/salida de reales:

```
package Float_InOut is new Float_IO(Float);
use Float_InOut;
```

Para escribir valores de tipo *Boolean*:

```
package Bool_InOut is new Enumeration_IO(Boolean);
use Bool_InOut;
```

Una vez creados estos paquetes puede utilizarse el procedimiento *Put* para escribirlos. Como nota diremos que en realidad, aunque todos ellos se llaman *Put*, para cada tipo de datos diferente estás usando un procedimiento *Put* diferente, específico para escribir el tipo de datos suministrado como argumento. A esto se le llama **sobrecarga** de nombres de función. Ada sabe distinguir procedimientos y funciones que, aunque tienen el mismo nombre, tienen distintos argumentos (otros lenguajes no).

Para utilizar funciones elementales matemáticas deberás crear una particularización del paquete de funciones elementales para el tipo de datos real. Este paquete se encuentra dentro de *Ada.Numerics*. Un ejemplo podría ser este:

```
1  with Ada.Numerics.Generic_Elementary_Functions;
2  use Ada.Numerics;

4  procedure MiPrograma is

6      package Math is new Generic_Elementary_Functions(Float);
7      use Math;
8      ...
```

```
10 begin
11     ...
12 end;
```

Por supuesto seguramente tengas que utilizar también *Text_IO* y particularizar paquetes para poder escribir el resultado de tus cálculos. Una vez particularizado el paquete *Generic_Elementary_Functions* dispondrás de las funciones más habituales de cálculo matemático (seno, coseno, etc.) Busca en Google qué funciones tiene este paquete. Prueba buscando ‘‘funciones matemáticas ada’’.

En este momento es normal recurrir a programas de ejemplo presentes en los apuntes y en los libros de texto para recordar cómo es la sintaxis del lenguaje. Si escribes los programas cada vez en lugar de cortar y pegar su código te será fácil recordar cómo hay que escribirlos.

Problemas

Escribe un programa en Ada que calcule cada uno de los siguientes problemas (Alguno de los enunciados corresponde a alguno de los problemas que ya hemos hecho, en tal caso hazlos tu de nuevo sin mirar cómo los hicimos antes).

1 Números de días de un año no bisiesto.

2 Volumen de una esfera.

3 Área de un rectángulo.

4 Área de un triángulo.

5 Volumen de un prisma de base triangular.

6 Área de un cuadrado.

7 Volumen de un prisma de base cuadrada.

8 Volumen de un prisma de base rectangular.

9 Área de una corona circular de radios interno y externo dados.

10 Volumen de un cilindro de radio dado al que se ha perforado un hueco vertical y cilíndrico de radio dado.

11 Primer dígito de la parte entera del número 14.5.

12 Raíz cuadrada de 2.

13 Valor absoluto de un número.

14 Factorial de 5.

15 Carácter siguiente a `'X'`.

16 Letras en el código ASCII entre las letras `'A'` y `'N'`.

17 Ver si un número entero tiene 5 dígitos.

18 Letra situada en la mitad del alfabeto mayúscula.

19 Media de los números 3, 5, 7 y 11.

20 Valor de la expresión

$$e^{2\pi}$$

21 Suma de los 10 primeros términos de la serie cuyos término general es

$$\left[1+\frac{1}{n}\right]^n$$

22 Convertir un ángulo de grados sexagesimales a radianes. Recuerda que hay 2π radianes en un círculo de 360^o.

23 Convertir de grados centígrados a farenheit una temperatura dada. Recuerda que

$$T_f = \frac{9}{5} T_c + 32$$

24 Ver si la expresión anterior (problema 21) es realmente el número *e*, cuyo valor es aproximadamente 2.71828.

25 Solución de la ecuación

$$x^2 + 15x - 3 = 0$$

26 Letra mayúscula correspondiente a la letra minúscula `'j'`.

27 Energía potencial de un cuerpo de masa conocida a una altura dada.

28 Valor de verdad que indique si un carácter es una letra minúscula.

29 Ver si un carácter es una letra.

30 Ver si un carácter es un signo de puntación (considerados como tales los que no son letras ni dígitos).

31 Valor de verdad que indique si un número es par o impar.

32 Número de segundos desde el comienzo del día hasta las 15:30:27 horas.

33 Hora, minutos y segundos correspondientes a los 9322 segundos desde el comienzo del día.

34 Ver si un número es mayor que otro.

35 Número de espacios que hay que imprimir en una línea de longitud dada la la izquierda de un texto para que dicho texto, de longitud también dada, aparezca centrado en dicha línea.

36 Ver si los números *a*, *b*, y *c* están ordenados de menor a mayor.

37 Carácter correspondiente al dígito `3`.

38 Dígito correspondiente al carácter `'8'`.

39 Número medio de semanas en un mes.

40 Ver si el primer carácter tiene la posición cero.

41 Número siguiente a uno dado en aritmética modular con módulo 10.

42 Menor número entero disponible.

43 Número de caracteres presentes en el juego de caracteres en el sistema.

44 Ver si hay más caracteres que enteros o viceversa.

45 Ver si se cumple que el dividendo es igual a la suma del resto y del producto del divisor y cociente.

46 Expresión de verdad para determinar si un año es bisiesto. Son bisiestos los años múltiplo de 4 salvo que sean múltiplos de 100. Excepto por que los múltiplos de 400 también son bisiestos.

47 Ver si la suma del cuadrado de un seno de un ángulo dado mas la suma del cuadrado de su coseno es igual a 1.

48 Ver si un número está en el intervalo [x,y).

49 Ver si un número está en el intervalo [x,y].

50 Ver si un carácter es un blanco (espacio en blanco, tabulador o fin de línea).

51 Dadas las coordenadas *x* e *y* del punto de arriba a la izquierda y de abajo a la derecha de un rectángulo y dadas las coordenadas de un punto, ver si el punto está dentro o fuera del rectángulo. En este problema hay que utilizar aritmética de números enteros y suponer que el origen de coordenadas está arriba a la izquierda. El eje de coordenadas horizontal es el eje *x* y el vertical es el eje *y*.

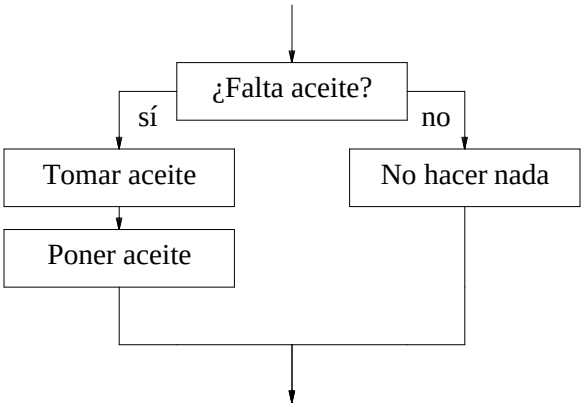
52 Ver si un número entero es un cuadrado perfecto (Esto es, es el cuadrado de algún número entero).

4 — Problemas de selección

4.1. Decisiones

No todos los problemas van a ser problemas que podemos solucionar directamente. Aunque hay muchísimos que lo son y que ya puedes programar en Ada, tal y como hemos hecho.

Si lo recuerdas, otra construcción típica que mencionamos para construir algoritmos es la selección. Recuerda esta figura:



La idea es que en ocasiones un problema requiere contemplar **distintos casos** y aplicar una solución distinta en cada caso. Hicimos esto cuando mencionamos el algoritmo para freír un huevo y lo vamos a volver a hacer durante el resto del curso.

Esto es similar a cuando en matemáticas se define una función de las llamadas ‘‘definidas a trozos’’; esto es, de las que corresponden a funciones distintas para distintos valores de entrada. Por ejemplo, el mayor de dos números se puede definir como el resultado de la función máximo:

$$maximo(a,b)=\begin{cases} a, & si\ a > b \\ b, & si\ a \leq b \end{cases}$$

Igualmente, podemos definir el signo de un número como la función que sigue:

$$signo(n)=\begin{cases} -1, & si\ n < 0 \\ 0, & si\ n = 0 \\ 1, & si\ n > 0 \end{cases}$$

Esto quiere decir que la función signo debe devolver -1 en unos casos, 0 en otros casos y 1 en otros casos. Igualmente, la función mayor deberá devolver *a* en unos casos y *b* en otros.

En Ada tenemos la sentencia (compuesta) llamada comúnmente *if-then-else* (o *si-entonces-sino*) para tomar decisiones y ejecutar unas u otras sentencias en función de que una condición se cumpla (sea cierta) o no (sea falsa). Por cierto, a esta y otras sentencias que modifican la ejecución secuencial habitual de un programa se las denomina **estructuras de control**. Volviendo al *if-then-else*, esta sentencia tiene la forma

```
if condición then      -- si ... entonces
       sentencias
else                   -- si no
       sentencias
end if;                -- fin si
```

Como con el resto de las sentencias, tenemos que terminarla con un ‘‘`;`’’. Cuando Ada tiene que

ejecutarla procede como sigue:

1 Se evalúa la condición, que ha de ser siempre una expresión de tipo *Boolean*. Esto es, una expresión que corresponde a un valor de verdad.

2 Si la condición se cumple (esto es, tiene como valor *True*) entonces se ejecutan las sentencias tras la palabra reservada *then* (llamadas ‘‘rama *then*’’). En otro caso se ejecutan las sentencias tras la palabra reservada *else* (llamadas ‘‘rama *else*’’). Tanto la rama *then* como la rama *else* son bloques de sentencias.

Bajo ninguna circunstancia se ejecutan ambas ramas. Y en cualquier caso, una vez terminado el *if-then-else*, se continúa como de costumbre con las sentencias que tengamos a continuación.

Veamos un ejemplo. La siguiente función devuelve el mayor de dos números.

```
1  function Maximo(a: Integer; b: Integer) return Integer is
2  begin
3      if a > b then
4             return a;
5      else
6             return b;
7      end if;
8  end;
```

En este punto merece la pena que vuelvas atrás y compares la definición de la función matemática definida a trozos con la función Ada programada arriba. Como podrás ver son prácticamente lo mismo, salvo por la forma de escribirlas. Fíjate además en que aquí sólo hay una condición, entre el *if* y el *then*. Cuando `a>b` es *False* no puede suceder otra cosa que que *b* sea el máximo.

Por cierto, debes prestar atención a la tabulación o sangrado que se utiliza al escribir sentencias *if-then-else*. Como verás, todas las sentencias dentro de la rama *then* están tabuladas más a la derecha; igual sucede con las de la rama *else*. Esto se hace para poner de manifiesto que dichas sentencias están *dentro* del *if-then-else*.

¿Y por qué debería importarme que estén dentro o fuera? Por que si están dentro, pero no te interesan ahora mismo, podrás ignorar de un sólo vistazo todas las sentencias que hay dentro y ver el *if-then-else* como una sóla cosa. No quieres tener en la cabeza todos los detalles del programa todo el tiempo. Interesa pensar sólo en aquella parte que nos importa, ignorando las demás. La vista humana está entrenada para reconocer diferencias de forma rápida (de otro modo se nos habrían comido los carnívoros hace mucho tiempo y ahora no estaríamos aquí y no podríamos programar). Tabulando los programas de esta forma graciosa tu cerebro identifica cada rama del *if-then-else* como una ‘‘cosa’’ o ‘‘mancha de tinta en el papel’’ distinta sin que tengas que hacerlo de forma consciente.

No siempre tendremos que ejecutar una acción u otra en función de la condición. En muchas ocasiones no será preciso hacer nada en uno de los casos. Para tales ocasiones tenemos la construcción llamada *if-then*, muy similar a la anterior:

```
if condición then      -- si ... entonces
       sentencias
end if;                -- fin si
```

Como podrás ver aquí sólo hay una rama. Si se cumple la condición se ejecutan las sentencias; si no se cumple se continúa ejecutando el código que sigue al *end if*. Por lo demás, esta sentencia es exactamente igual a la anterior.

Para ver otro ejemplo, podemos programar nuestra función *Signo* en Ada como sigue:

```
1  function Signo(n: Integer) return Integer is
2  begin
3      if n > 0 then
4             return 1;
5      else
6             if n = 0 then
7                    return 0;
8             else
9                    return -1;
10            end if;
11     end if;
12 end;
```

En este caso teníamos que distinguir entre tres casos distintos según el número fuese mayor que cero, cero o menor que cero. Pero el *if-then-else* es binario y sólo sabe distinguir entre dos casos. Lo que hacemos es ir considerando los casos a base tomar decisiones de tipo ‘‘sí o no’’: si es mayor que cero, entonces el valor es 1; en otro caso ya veremos lo que hacemos. Si pensamos ahora en ese otro caso, puede que *n* sea cero o puede que no. Y no hay más casos posibles.

4.2. Múltiples casos

En algunas situaciones es posible que haya que considerar muchos casos, pero todos mutuamente excluyentes. Esto es, que nunca podrán cumplirse a la vez las condiciones para dos casos distintos. La función *Signo* es un ejemplo de uno de estos casos. En estas situaciones podemos utilizar una variante de la sentencia *if-then-else*, llamada *if-then-elsif*, que permite encadenar múltiples *if-then-elses* de una forma más cómoda. Esta sentencia tiene el aspecto

```
    if condición then      -- si ... entonces
           sentencias
    elsif condición then   -- si no, si ... entonces
           sentencias
    elsif condición then   -- si no, si ... entonces
           sentencias
    else                   -- si no
           sentencias
    end if;                -- fin si
```

Esto es sólo ayuda sintáctica, para que no tengamos que poner muchos *end-if* al final de todos los *ifs*. Pero es cómodo. Esta es la función `Signo` programada de un modo un poco mejor:

```
1  function Signo(n: Integer) return Integer is
2  begin
3      if n > 0 then
4             return 1;
5      elsif n = 0 then
6             return 0;
7      else
8             return -1;
9  end;
```

Hasta ahora hemos utilizado los *if-then* para definir funciones en Ada. Pero en realidad se utilizan para ejecutar sentencias de forma condicional; cualquier sentencia. Dicho de otro modo: no se trata de poner *returns* rodeados de *if-thens*. Por ejemplo, este código imprime el valor de *a*/*b*, pero tiene mucho cuidado de no dividir por 0 (lo que provocaría un error y la parada del programa en la mayoría de los lenguajes de programación).

```
if b /= 0 then
       Put(a / b);
else
       Put("No puedo. b = 0");
end if;
```

Aunque claro, quizá otra opción es escribir el equivalente a Infinito en tal caso:

```
if b /= 0 then
       Put(a / b);
else
       Put(Integer'Last);
end if;
```

Hay una tercera estructura de control para ejecución condicional de sentencias. Esta estructura se denomina *case* y está pensada para situaciones en que discriminamos un caso de otro en función de un valor discreto perteneciente a un conjunto (un valor de tipo *Integer* o de tipo *Character*, por ejemplo). Su uso está indicado cuando hay múltiples casos excluyentes, de ahí su nombre. La estructura de la sentencia *case* es como sigue:

```
case expresión is
when valores =>
       sentencias
when valores =>
       sentencias
when valores =>
       sentencias
 ...
when others =>
       sentencias
end case;
```

De nuevo, utilizamos un ‘‘;’’ al final. Ada evalúa la expresión primero y luego ejecuta las sentencias de la rama correspondiente al valor que adopta la expresión. Si la expresión no toma como valor ninguno de los valores indicados tras *when* entonces se ejecuta la rama *when others*. Por último, igual que de costumbre, la ejecución continúa con las sentencias escritas tras el *end case*.

Por ejemplo, esta función devuelve como resultado el valor de un dígito hexadecimal. En base 16, llamada hexadecimal, se utilizan los dígitos del 0 al 9 normalmente y se utilizan las letras de la *A* a la *F* para representar los dígitos con valores del 10 al 15. Esta función soluciona el problema por casos, seleccionando el caso en función del valor de *digito*.

```
1  function ValorDigito16(digito: Character) return Integer is
2  begin
3      case digito is
4      when '0' =>
5             return 0;
6      when '1' =>
7             return 1;
8      when '2' | '3' =>
9             return Character'Pos(digito) - Character'Pos('0');
10     when 'A'..'F' =>
11            return Character'Pos(digito) - Character'Pos('A') + 10;
12     when others =>
13            return 0;
14     end case;
15 end;
```

Cuando Ada empieza a ejecutar el *case* lo primero que hace es evaluar *digito*. Luego se ejecuta una rama u otra en función del valor del dígito. La rama

```
4         when '0' =>
5                 return 0;
```

hace que la función devuelva 0 cuando *digito* es `'0'`. La segunda rama es similar, pero cubre el caso en que el dígito es el carácter `'1'`.

La rama

```
8         when '2' | '3' =>
9                 return Character'Pos(digito) - Character'Pos('0');
```

muestra una forma de ejecutar la misma rama para varios valores distintos. En este caso, cuando *digito* sea `'2'` o `'3'` se ejecutará esta rama.

La rama

```
10        when 'A'..'F' =>
11                return Character'Pos(digito) - Character'Pos('A') + 10;
```

muestra otra forma de ejecutar una misma rama para múltiples valores consecutivos de la expresión usada como selector del caso. Esta resulta útil cuando hay que hacer lo mismo para múltiples valores y todos ellos son consecutivos. En este caso, para las letras de la `'A'` a la `'F'` hay que calcular el valor del mismo modo.

Una sentencia *case* debe obligatoriamente cubrir todos los valores posibles del tipo de datos al que pertenece el valor que discrimina un caso de otro. Esto quiere decir que en la práctica hay que incluir una rama *when others* en la mayoría de los casos.

Nótese que la función se habría podido escribir de un modo más compacto utilizando

```
case digito is
when '0'..'9' =>
    return Character'Pos(digito) - Character'Pos('0');
when 'A'..'F' =>
    return 10 + Character'Pos(digito) - Character'Pos('A');
when others =>
    return 0;     -- hay un error en el programa.
end case;
```

pero hemos querido aprovechar para mostrar las formas típicas de seleccionar los distintos casos.

Las expresiones *Valor1*`..`*Valor2* se denominan **rangos**. Representan el subconjunto de valores contenidos entre *Valor1* y *Valor2*, incluidos ambos. Resultan muy útiles en las sentencias *case*.

4.3. Punto más distante a un origen

Supongamos que tenemos valores discretos dispuestos en una línea y tomamos uno de ellos como el origen. Nos puede interesar ver cuál de dos valores es el más distante al origen. Por ejemplo, en la figura 4.1, ¿Será el punto *a* o el *b* el más lejano al origen *o*?

Figura 4.1: *Dos puntos a y b en una recta discreta con origen en o.*

Procedemos como de costumbre. Si suponemos que tenemos una función *DistanciaLineal* que da la distancia entre dos puntos, y una constante *Origen* que corresponde a *o* en la figura, entonces podemos programar una solución como sigue:

```
1  -- posicion del punto mas distante al origen en una recta.
2  function MasDistante(a: Integer; b: Integer) return Integer is
3  begin
4      if DistanciaLineal(Origen, a) > DistanciaLineal(Origen, b) then
5             return a;
6      else
7             return b;
8      end if;
9  end;
```

Naturalmente, necesitamos programar la función *DistanciaLineal* para completar el programa. Esta función debe tener como valor la diferencia entre ambos puntos, pero en valor absoluto; cosa que podemos programar como sigue:

```
1  -- distancia entre pos puntos en una recta discreta.
2  function DistanciaLineal(a: Integer; b: Integer) return Integer is
3  begin
4      if a > b then
5             return a - b;
6      else
7             return b - a;
8      end if;
9  end;
```

Por otra parte, una vez teníamos programada la función *Signo* vista antes, podríamos haber programado esta función como

```
1  -- distancia entre pos puntos en una recta.
2  function DistanciaLineal(a: Integer; b: Integer) return Integer is
3  begin
4      return (a - b) * Signo(a - b);
5  end;
```

dado que al multiplicar la diferencia por su signo obtenemos la diferencia en valor absoluto. Pero parece mucho más directa y sencilla la primera implementación.

El programa completo quedaría como sigue. Como podrá verse, su implementación es un poco diferente, para mostrar una tercera versión.

distancia.adb

```
1      --
2      -- Programa para calcular distancias.

4      with Ada.Text_IO;
5      use Ada.Text_IO;

7      procedure Distancia is

9           -- Necesario para escribir Integers.
10          package Integer_InOut is new Integer_IO(Integer);
11          use Integer_InOut;

13          -- Ponemos nuestro punto origen en el 2.
14          Origen: constant := 2;
```

```
16          -- -1, 0 o 1 para un numero negativo, cero o positivo.
17          function Signo(n: Integer) return Integer is
18          begin
19               if n > 0 then
20                    return 1;
21               elsif n < 0 then
22                    return -1;
23               else
24                    return 0;
25               end if;
26          end;

28          -- valor absoluto de un entero.
29          function ValAbs(n: Integer) return Integer is
30          begin
31               return n * Signo(n);
32          end;

34          -- distancia entre pos puntos en una recta discreta.
35          function DistanciaLineal(a: Integer; b: Integer) return Integer is
36          begin
37               return ValAbs(a - b);
38          end;

40          -- posicion del punto mas distante al origen en una recta.
41          function MasDistante(a: Integer; b: Integer) return Integer is
42          begin
43               if DistanciaLineal(Origen, a) > DistanciaLineal(Origen, b) then
44                    return a;
45               else
46                    return b;
47               end if;
48          end;

50     --
51     -- Constantes de pruebas.
52     --
53          Prueba1: constant Integer := MasDistante(2, -3);
54          Prueba2: constant Integer := MasDistante(-2, -3);

56     begin
57          -- Ejecucion de pruebas.
58          Put(Prueba1);
59          New_Line;
60          Put(Prueba2);
61          New_Line;
62     end;
—
```

4.4. Mejoras

Hay varios casos en los que frecuentemente se escribe código que es manifiestamente mejorable. Es importante verlos antes de continuar.

En muchas ocasiones todo el propósito de un *if-then-else* es devolver un valor de verdad, como en

```
function EsPositivo(n: Integer) return Boolean is
begin
    if n > 0 then
           return True;
    else
           return False;
    end if;
end;
```

Pero hacer esto no tiene mucho sentido. La condición expresa ya el valor que nos interesa. Aunque sea de tipo *Boolean* sigue siendo un valor. Podríamos entonces hacerlo mucho mejor:

```
function EsPositivo(n: Integer) return Boolean is
begin
return n > 0;
end;
```

Ten esto presente. Dado que los programas se hacen poco a poco es fácil acabar escribiendo código demasiado ineficaz. No pasa nada; pero en cuanto lo vemos lo mejoramos.

Otro caso típico es aquel en que se realiza la misma acción en todas las ramas de una estructura de ejecución condicional. Por ejemplo,

```
if a > b then
    Put("el numero");
    Put(a);
    Put("es mayor que");
    Put(b);
    New_Line;
else
    Put("el numero");
    Put(a);
    Put("es menor o igual que");
    Put(b);
    New_Line;
end if;
```

Mirando el código vemos que tenemos las mismas sentencias al principio de ambas ramas. Es mucho mejor sacar ese código del *if-then-else* de tal forma que tengamos una única copia de las mismas:

```
Put("el numero");
Put(a);
if a > b then
    Put("es mayor que");
    Put(b);
    New_Line;
else
    Put("es menor o igual que");
    Put(b);
    New_Line;
end if;
```

Si lo miramos de nuevo, veremos que pasa lo mismo al final de cada rama. Las sentencias son las mismas. En tal caso volvemos a aplicar la misma idea y sacamos las sentencias fuera del *if-then-else*.

```
1  Put("el numero");
2  Put(a);
3  if a > b then
4      Put("es mayor que");
5  else
6      Put("es menor o igual que");
7  end if;
8  Put(b);
9  New_Line;
```

¿Y por qué importa esto? Principalmente, por que si comentemos un error en las sentencias que estaban repetidas es muy posible que luego lo arreglemos sólo en una de las ramas y nos pase la otra inadvertida. Además, así ahorramos código y conseguimos un programa más sencillo, que consume menos memoria en el ordenador. Hay una tercera ventaja: el código muestra con mayor claridad la diferencia entre una rama y otra.

4.4.1. Primeros tres dígitos hexadecimales

Queremos escribir los primeros tres dígitos en base 16 para un número dado en base 10. En base 16 los dígitos son *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E* y *F*; utilizando letras de la *A* a la *F* para representar dígitos con valores 10, 11, 12, 13, 14 y 15, respectivamente.

Volvemos a aplicar el mismo sistema que hemos estado utilizando ya desde hace un tiempo. En realidad necesitamos dos cosas: obtener el valor de cada dígito y obtener el carácter correspondiente a un dígito. Una vez tengamos esto, podemos realizar nuestro programa como sigue:

```
1  Put( CarDigito16( ValorDigito(Numero, 3, 16) ) );
2  Put( CarDigito16( ValorDigito(Numero, 2, 16) ) );
3  Put( CarDigito16( ValorDigito(Numero, 1, 16) ) );
```

Vamos a utilizar *ValorDigito* ya que lo tenemos de un problema anterior (¿Ves como resulta útil generalizar siempre que cueste el mismo esfuerzo? Ahora no tenemos que hacer esta función, la que teníamos también funciona para este problema puesto que optamos por considerar también la base de numeración. Esta vez la base es 16 y no 10.

Y nos vamos a inventar la función *CarDigito16*, ‘‘carácter para un dígito en base 16’’, que nos devolverá el carácter para el dígito en base 16 cuyo valor se le suministra. Luego... ¡Lo tenemos hecho! Basta programar esta última función.

Primero definir el problema: suministramos un entero y queremos un carácter que corresponda con el dígito. Luego la cabecera de la función ha de ser:

```
function CarDigito16(valor: Integer) return Character
```

Esta función es un caso típico de un problema de casos. Si el valor está entre 0 y 9 entonces podemos sumar el valor a la posición que ocupa el `'0'` en el código de caracteres y de esta forma obtener el código para el carácter correspondiente. Convirtiendo ese código a un carácter tenemos la función resuelta. En otro caso, el valor ha de estar entre 10 y 15 y lo que tenemos que hacer es lo mismo pero sumando el valor al código del carácter `'A'`.

```
1  -- Caracter para un digito en base 16.
2  function CarDigito16(valor: Integer) return Character is
3  begin
4      if valor >= 0 and valor <= 9 then
5             return Character'Val(Character'Pos('0') + valor);
6      else
7             return Character'Val(Character'Pos('A') + valor);
8      end if;
9  end;
```

4.5. ¿Es válida una fecha?

Tenemos una fecha (quizá escrita por el usuario en un teclado) y queremos estar seguros de que la fecha es válida.

Claramente tenemos tres subproblemas aquí: ver si un año es válido, ver si un mes es válido y ver si un día es válido. El primer subproblema y el segundo parecen sencillos y, suponiendo que el tercero lo tenemos ya programado, podemos escribir el programa entero como sigue:

```
1  --
2  -- Es valida una fecha?
3  --

5  with Ada.Text_IO;
6  use Ada.Text_IO;

8  procedure FechaOk is

10     -- Necesario para escribir Booleans.
11     package BIO is new Enumeration_IO(Boolean);
12     use BIO;



15     function EsAnnoValido(anno: Integer) return Boolean is
16     begin
17            return anno >= 1 and anno < 3000;
18     end;

20     function EsMesValido(mes: Integer) return Boolean is
21     begin
22            return mes >= 1 and mes <= 12;
23     end;

25     --  Falta programar esta funcion !!
26     function EsDiaValido(dia: Integer) return Boolean is
27     begin
28            return True;
29     end;

31     function EsFechaValida(dia: Integer;
32                   mes: Integer;
33                   anno: Integer) return Boolean is
34     begin
35            return EsAnnoValido(anno) and EsMesValido(mes) and EsDiaValido(dia);
36     end;

38     -- Constantes de pruebas
39     Prueba1: constant Boolean := EsFechaValida(1,1, 1970);
40     Prueba2: constant Boolean := EsFechaValida(29,2, 1970);

42 begin
43     Put(Prueba1);
44     Put(Prueba2);
45 end;
```

Ahora hay que pensar cómo vemos si el día es válido. En cuanto lo hagamos, veremos que hemos cometido el error de suponer que podemos calcular tal cosa sólo pensando en el valor del día. Necesitamos también tener en cuenta el mes y en año (¡debido a febrero!). Luego la cabecera de la función va a ser

```
function EsDiaValido(dia: Integer;
             mes: Integer;
             anno: Integer) return Boolean
```

y tendremos que cambiar la función *EsFechaValida* para que pase los tres argumentos en lugar de sólo el día.

La función *EsDiaValido* parece por lo demás un caso típico de problema por casos. Hay meses que tienen 30 y hay meses que 31. El caso de febrero va a ser un poco más complicado. Parece una función a la medida de un *case*, dado que según el valor del mes tenemos un caso u otro de entre 12 casos distintos.

```
1  function EsBisiesto(anno: Integer) return Boolean is
2  begin
3      return anno mod 4 = 0 and
4             (not (anno mod 100 = 0) or anno mod 400 = 0);
5  end;

7  function EsDiaValido(dia: Integer;
8                   mes: Integer;
9                   anno: Integer) return Boolean is
10 begin
11     case mes is
12     when 1|3|5|7|8|10|12 =>
13            return dia >= 1 and dia <= 31;
14     when 4|6|9|11 =>
15            return dia >= 1 and dia <= 30;
16     when 2 =>
17            if EsBisiesto(anno) then
18                   return dia >= 1 and dia <= 29;
19            else
20                   return dia >= 1 and dia <= 28;
21            end if;
22     when others =>
23            return False;
24     end case;
25 end;
```

No obstante, mirando *EsDiaValido* vemos que estamos repitiendo muchas veces lo mismo: comparando si un día está entre 1 y el número máximo de días. Viendo esto podemos mejorar esta función haciendo que haga justo eso y sacando el cálculo del número máximo de días del mes en otra nueva función.

El programa completo queda como sigue.

fechaok.adb

```
1      --
2      -- Es valida una fecha?
3      --

5      with Ada.Text_IO;
6      use Ada.Text_IO;

8      procedure FechaOk is

10         -- Necesario para escribir Booleans.
11         package BIO is new Enumeration_IO(Boolean);
12         use BIO;
```

```
15        function EsAnnoValido(anno: Integer) return Boolean is
16        begin
17             return anno >= 1 and anno < 3000;
18        end;

20        function EsMesValido(mes: Integer) return Boolean is
21        begin
22             return mes >= 1 and mes <= 12;
23        end;

25        function EsBisiesto(anno: Integer) return Boolean is
26        begin
27             return anno mod 4 = 0 and
28                  (not (anno mod 100 = 0) or anno mod 400 = 0);
29        end;

31        function MaxDiaMes(mes: Integer; anno: Integer) return Integer is
32        begin
33             case mes is
34             when 1|3|5|7|8|10|12 =>
35                  return 31;
36             when 4|6|9|11 =>
37                  return 30;
38             when 2 =>
39                  if EsBisiesto(anno) then
40                          return 29;
41                  else
42                          return 28;
43                  end if;
44             when others =>
45                  return 0;
46             end case;
47        end;

49        function EsDiaValido(dia: Integer;
50                        mes: Integer;
51                        anno: Integer) return Boolean is
52        begin
53             return dia >= 1 and dia <= MaxDiaMes(mes, anno);
54        end;

56        function EsFechaValida(dia: Integer;
57                        mes: Integer;
58                        anno: Integer) return Boolean is
59        begin
60             return EsAnnoValido(anno) and EsMesValido(mes) and
61                  EsDiaValido(dia, mes, anno);
62        end;

64        -- Constantes de pruebas
65        Prueba1: constant Boolean := EsFechaValida(1,1, 1970);
66        Prueba2: constant Boolean := EsFechaValida(29,2, 1970);
```

```
69    begin
70          Put(Prueba1);
71          Put(Prueba2);
72    end;
—
```

En general, refinar el programa cuando es sencillo hacerlo suele compensar siempre. Ahora tenemos no sólo una función que dice si un día es válido o no. También tenemos una función que nos dice el número de días en un mes. Posiblemente sea útil en el futuro y nos podamos ahorrar hacerla de nuevo.

Problemas

Escribe un programa en Ada que calcule cada uno de los siguientes problemas (Para aquellos enunciados que corresponden a problemas ya hechos te sugerimos que los hagas de nuevo pero esta vez sin mirar la solución).

1 Valor absoluto de un número.

2 Signo de un número. El signo es una función definida como

$$signo(x) = \begin{cases} -1, \textit{ si x es negativo} \\ 0, \textit{ si x es cero} \\ 1, \textit{ si x es positivo} \end{cases}$$

3 Valor numérico de un dígito hexadecimal.

4 Distancia entre dos puntos en una recta discreta.

5 Ver si número entero es positivo.

6 Máximo de dos números enteros.

7 Máximo de tres números enteros

8 Convertir un carácter a mayúscula, dejándolo como está si no es una letra minúscula.

9 Devolver el número de círculos que tiene un dígito (gráficamente al escribirlo: por ejemplo, 0 tiene un círculo, 2 no tiene ninguno y 8 tiene dos).

10 Indicar el cuadrante en que se encuentra un número complejo dada su parte real e imaginaria.

11 Devolver un valor de verdad que indique si un número de carta (de 0 a 10) corresponde a una figura.

12 Devolver el valor de una carta en el juego de las 7 y media.

13 Devolver el número de días del mes teniendo en cuenta si el año es bisiesto.

14 Indicar si un número de mes es válido o no.

15 Indicar si una fecha es válida o no.

16 Ver si un número puede ser la longitud de la hipotenusa de un triángulo rectángulo de lados dados.

17 Decidir si la intersección de dos rectángulos es vacía o no.

18 Decidir si se aprueba una asignatura dadas las notas de teoría y práctica teniendo en cuenta que hay que aprobarlas por separado (con notas de 0 a 5 cada una) pero pensando que se considera que un 4.5 es un 5 en realidad.

19 Devolver el primer número impar mayor o igual a uno dado.

20 Devolver el número de días desde comienzo de año dada la fecha actual.

21 La siguiente letra a una dada pero suponiendo que las letras son circulares (esto es, detrás de la `'z'` vendría de nuevo la `'a'`).

22 Resolver una ecuación de segundo grado dados los coeficientes, sean las raíces de la ecuación reales o imaginarias.

23 Determinar si tres puntos del plano forman un triángulo. Recuerda que la condición para que tres puntos puedan formar un triángulo es que se cumpla la condición de triangularidad. Esto es, que cada una de las distancias entre dos de ellos sea estrictamente menor que las distancias correspondientes a los otros dos posibles lados del triángulo.

5 — Acciones y procedimientos

5.1. Efectos laterales

Hasta el momento hemos utilizado constantes y funciones para realizar nuestros programas. Si programamos limpiamente, ambas cosas se comportarán como funciones para nosotros. Una constante es simplemente una función que no recibe argumentos y, como tal, siempre devuelve el mismo valor.

En este capítulo vamos a cambiar todo esto. Aunque parezca que a lo largo del capítulo estamos hablando de cosas muy diferentes, en realidad estaremos siempre hablando de acciones y variables.

Si nuestras funciones sólo utilizan sus parámetros como punto de partida para elaborar su resultado y no consultan ninguna fuente exterior de información ni producen ningún cambio externo en el programa (salvo devolver el valor que deban devolver), entonces las funciones se comportan de un modo muy similar a una función matemática (o a una constante, salvo por que el valor depende de los argumentos).

A esto se le denomina **transparencia referencial**. Esto es: cambiar en un programa una función y sus argumentos por el valor que produce esa función con esos argumentos no altera el resultado del programa.

Bastaría utilizar una sentencia *Put* dentro de una función para romper la transparencia referencial. Por ejemplo, si la función *Media* imprime un mensaje en lugar (o además) de devolver el valor correspondiente a la media de dos números entonces ya no tendrá transparencia referencial. Ello es debido a que la función (con una llamada a *Put*) produciría un efecto visible de forma externa (el mensaje). A eso se le denomina **efecto lateral** (Ver figura 5.1).

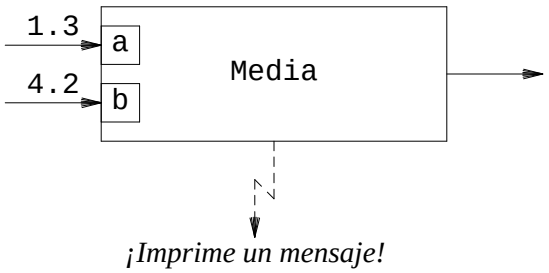


Figura 5.1: *En un efecto lateral una función provoca un cambio visible desde fuera.*

Lo que importa de esto es que si tenemos una función que produce efectos laterales (hace cosas que no debe) entonces *no* podremos olvidarnos del efecto lateral. Por ejemplo, si *Media* escribe un mensaje, entonces no es igual llamar a esta función una que tres veces. ¡Pero debería serlo! Un fragmento de código como

```
if Media(a, b) = 2 then
      ...
end if;
```

no parece imprimir ningún mensaje y, desde luego, no parece que cambie su comportamiento si volvemos a comprobar si la media es 3, por ejemplo, como en

```
if Media(a, b) = 2 then
       ...
end if;
if Media(a, b) = 3 then
       ...
end if;
```

El precio que pagamos por poner un efecto lateral es que tendremos que estar siempre pensando en qué hace dentro *Media*, en lugar de olvidarnos por completo de ello y pensar sólo en lo que puede leerse viendo ‘‘`Media(a,b)`’’.

No obstante, hay otro tipo de subprograma llamado **procedimiento**, o *procedure*, pensado justamente para producir efectos laterales. Esto es bueno, puesto que de otro modo todos nuestros programas serían autistas. Realmente ya los conocemos: *Put* es un procedimiento.

Igual sucede con las constantes. Además de las constantes tenemos otro tipo de entidades capaces de mantener un valor: las **variables**. Como su nombre indica, una variable es similar a una constante salvo por que podemos hacer que su valor varíe.

Al estilo de programación que hemos mantenido hasta el momento se lo denomina **programación funcional** (que es un tipo de programación **declarativa**). En este estilo estamos más preocupados por definir las cosas que por indicar cómo queremos que se hagan. Introducir procedimientos y variables hace que hablemos que **programación imperativa**, que es otro estilo de programación. En éste estamos más preocupados por indicar cómo queremos que se hagan las cosas que por definir funciones que lo hagan.

5.2. Variables

Un programa debe ser capaz de hacer cosas distintas cada vez que lo ejecutamos. Si todos los programas estuviesen compuestos sólo de funciones y constantes esto no sería posible.

El primer elemento que necesitamos es el concepto de **variable**. *Una variable es en realidad un nombre para un objeto* o para un valor, si se prefiere ver así. Podemos crear cuantas variables queramos. Para crear una variable hay que declararla indicando claramente su nombre (su identificador) y el tipo de datos para el valor que contiene. Por ejemplo,

```
numero: Integer;
```

declara una variable llamada *numero* de tipo *Integer*. A partir del momento de la declaración, y hasta que termine la ejecución de la función o procedimiento donde se ha declarado la variable, dicha variable existirá como una cantidad determinada de espacio reservado en la memoria del ordenador a la que podemos referirnos simplemente mediante el nombre que le hemos dado (ver figura 5.2).

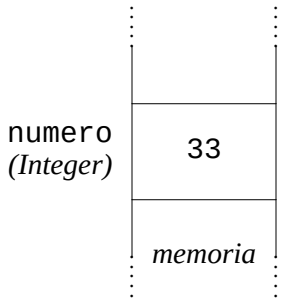


Figura 5.2: *Una variable es un nombre para una zona de memoria que guarda un valor de un tipo.*

Como nombres de variable vamos a utilizar identificadores que están escritos en minúsculas, para distinguirlos de constantes y funciones. Es preciso utilizar **nombres cortos que resulten obvios** al verlos. Conviene que los nombres no sean muy largos para que no tengamos sentencias farragosas y difíciles de leer, pero tampoco hay que escatimar caracteres. El nombre más corto que indique de forma evidente a lo que se refiere es el mejor nombre posible.

Por ejemplo, si tenemos una variable en un subprograma que calcula el área de un círculo podríamos utilizar *r* como nombre para el radio, dado que resulta obvio lo que es *r* al verlo en una expresión como ‘‘`Pi*r**2`’’. En este caso sería pésimo un nombre como *valordelradiodelcirculo* dado que es mucho más largo de lo necesario y hará que las expresiones dónde usemos la variable queden largas y pesadas. En cambio, si tenemos variables para el valor de una carta y el palo (de la baraja) de la misma en un subprograma, sería desastroso utilizar nombres como *v* y *p* para el valor y el palo respectivamente. No se puede esperar que nadie sepa qué es *v* y qué es *p*. Lo lógico sería llamarles *valor* y *palo*.

Si en algún momento necesitamos variables para el número de peras en una cesta y para el número de manzanas en una cesta, podríamos declarar variables con nombres *numperas* y *nummanzanas*, que son claros. O tal vez *nperas* y *nmanzanas*. Si usamos nombres como *n1* y *n2* o bien *np* y *nm* el código será ilegible. Si utilizamos nombres como *numerodeperasenlacesta* y *numerodemanzanasenlacesta* es muy posible que las sentencias sean también ilegibles (de lo largas que resultarán).

Si tu compañero no sabe lo que son tus variables entonces tus nombres están mal. Si tus nombres son tan largos que el código no te cabe en el editor y tienes que ensanchar la ventana entonces tus nombres también están mal. Si tienes tus nombres bien te resultará muy fácil leer tu programa y será mucho más difícil que cometas errores al usarlos.

5.3. Asignación

Además de declararlas, hay dos operaciones básicas que podemos hacer con una variable: consultar su valor y cambiar su valor. Para consultar el valor de la variable basta con usar su nombre. Por ejemplo, si queremos escribir por la salida estándar el valor de la variable *numero* podemos escribir esta sentencia:

```
Put(numero);
```

Dicha sentencia **consulta** el valor de la variable *numero* y se lo pasa como argumento a *Put*. Igualmente, si escribimos la expresión ‘‘`Pi*r**2`’’ entonces Ada consulta el valor de *r* para calcular el valor de la expresión.

Para cambiar el valor que tiene una variable disponemos de una sentencia denominada **asignación**, que tiene la forma:

```
variable := valor;
```

Esta sentencia evalúa primero la expresión escrita a la derecha de ‘‘`:=`’’ y posteriormente guarda dicho valor en la variable cuyo nombre está a la izquierda de ‘‘`:=`’’. Por ejemplo,

```
numero := 3;
```

cambia el valor de la variable *numero* y hace que en la memoria utilizada para almacenar dicha variable se guarde el valor 3, de tipo *Integer*, escrito a la derecha del operador ‘‘`:=`’’.

A la izquierda de ‘‘`:=`’’ sólo pueden escribirse nombres de variable, dado que sólo tiene sentido utilizar la asignación para asignarle (darle) un valor a una variable.

A la derecha de ‘‘`:=`’’ sólo pueden escribirse expresiones cuyo valor sea del tipo de datos al que pertenece la variable. Esto es, una variable siempre tiene el mismo tipo de datos desde que nace (se declara) hasta que muere (su subprograma acaba).

Veamos lo que sucede si ejecutamos un programa que declara y utiliza una variable.

```
1  procedure UsandoVariables is
2      numero: Integer;
3  begin
4      Put("Voy a utilizar una variable");
5      New_Line;
6      numero := 3;
7      Put("Le he dado un valor y vale ");
8      Put(numero);
9      New_Line;
10 end;
```

Al ejecutar el programa se procesa primero la zona de declaraciones entre *procedure* y *begin*. En este punto, al alcanzar la línea 2, se declara la variable *numero* y se reserva espacio en la memoria para mantener un entero. Una vez procesadas todas las declaraciones, Ada ejecuta las sentencias del cuerpo del programa (líneas 3 a 10) hasta alcanzar el ‘‘`end;`’’. En este punto la variable deja de existir dado que el (sub)programa donde se ha declarado ha dejado de existir.

Entre la línea 3 y la línea 6 *no sabemos cuanto vale la variable*. Esto quiere decir que *no podemos utilizar la variable* en ninguna expresión puesto que aún no le hemos dado un valor y el valor que tenga será lo que hubiese en la memoria del ordenador allí donde se le ha dado espacio a la variable. ¡Y no sabemos cuál es ese valor! Utilizar la variable antes de asignarle un valor tiene efectos impredecibles.

La línea 6 **inicializa** la variable. Esto es, le asigna un valor inicial. En este caso el valor 3 de tipo *Integer*. A partir de este momento la variable guarda el valor 3 hasta que se le asigne alguna otra cosa.

La línea 8 consulta la variable como parte de la expresión *Put(numero)*, que escribe el valor de la variable en la salida estándar.

Veamos otro ejemplo. Supongamos que tenemos el programa:

```
1  procedure MasVariables is
2      x: Integer;
3      y: Integer;
4  begin
5      x := 3;
6      y := x + 1;
7      x := y / 2;
8  end;
```

La primera sentencia del cuerpo (línea 5) inicializa *x* a *3*. Antes de esta línea no sabemos ni cuánto vale *x* ni cuánto vale *y*; por lo que no deberíamos usarlas a la derecha de una asignación o como parte de una expresión. Pasada la línea 5 *x* está inicializada, pero no así *y*. La línea 6 evalúa ‘‘`x + 1`’’ (que tiene como valor 4, dado que *x* vale 3) y asigna este valor a la variable *y*, que pasa a valer 4. Igualmente, la línea 7 evalúa ‘‘`y / 2`’’ (cuyo valor es 2, dado que *y* vale 4) y asigna ese valor a *x*. El programa termina con *x* valiendo 2 y con *y* valiendo 4.

La siguiente secuencia de sentencias es interesante:

```
x := 3;
x := x + 1;
```

En la primera línea se le da el valor 3 a la variable *x*. En la segunda línea se utiliza *x* tanto en la parte derecha como en la parte izquierda de una asignación. Esto es perfectamente válido. *La asignación no es una ecuación matemática*. La asignación da un nuevo valor a una variable. De hecho, no hay que confundir nuestra sentencia con

```
x = x + 1
```

que es una expresión de tipo *Boolean* cuyo valor es *False* (puesto que *x* sólo puede tener un único

valor a la vez).

Pero veamos lo que hace

```
x := x + 1;
```

Primero Ada evalúa la parte derecha de la asignación: ‘‘`x + 1`’’. Esta es una expresión que calcula el valor de una suma de una variable *x* de tipo *Integer* y de un literal ‘‘`1`’’ de tipo *Integer*. El valor de la expresión es el resultado de sumar el valor de la variable y 1. Esto es, 4 en este ejemplo.

A continuación, el valor calculado para la parte derecha de la asignación se le asigna a la variable escrita en la parte izquierda de la asignación. Como resultado, *x* pasa a tener el valor 4. Esto es,

```
x := x + 1;
```

ha incrementado el valor de la variable en una unidad. Si antes de la asignación valía 3 después valdrá 4. A esta sentencia se la conoce como **incremento**, por no se sabe qué oscuras razones. Es una sentencia popular en variables destinadas a contar cosas (llamadas **contadores**).

Una variable debería representar algún tipo de magnitud o entidad real y tener un valor que corresponda con esa entidad. Por ejemplo, si hacemos un programa para jugar a las cartas y tenemos una variable *numcartas* que representa el número de cartas que tenemos en la mano, esa variable siempre debería tener como valor el número de cartas que tenemos en la mano, sin importar la línea del programa en la que consultemos la variable. De no hacerlo así, es fácil confundirse.

5.4. Más sobre variables

Podemos tener variables de cualquier tipo de datos. Por ejemplo:

```
c: Character;

 ...

c := 'A';
```

declara la variable *c* de tipo *Character* y le asigna el carácter cuyo literal es `'A'`. A partir de este momento podríamos escribir una sentencia como

```
c := Character'Val(Character'Pos(c) + 1);
```

que haría que *c* pasase a tener el siguiente carácter en el código ASCII. Esto es, `'B'`. Esto es así puesto que *c* tenía inicialmente el valor `'A'`, con lo que

```
Character'Pos(c)
```

es el número que corresponde a la posición de `'A'` en el juego de caracteres. Si sumamos una unidad a dicho número tenemos la siguiente posición en el juego de caracteres. Si ahora utilizamos *Character'Val* para obtener el carácter con dicha posición, obtenemos el valor `'B'` de tipo *Character*. Este valor se lo hemos asignado a *c*, con lo que *c* pasa a valer `'B'`. Una forma más directa habría sido utilizar

```
c := Character'Succ(c);
```

que hace que *c* pase a tener el siguiente carácter de los presentes en el tipo *Character*.

En muchas ocasiones es útil utilizar una variable para almacenar un valor que el usuario escribe en la entrada estándar del programa. Así podemos hacer que nuestro programa lea datos de la entrada y calcule una u otra cosa en función de dichos datos.

Podemos pues inicializar una variable dándole un valor que leemos de la entrada el programa. Normalmente decimos que en tal caso **leemos** la variable de la entrada del programa (En realidad leemos un valor que se asigna a la variable). Esta acción puede realizarse llamando al procedimiento *Get*, del paquete *Text_IO*. Por ejemplo, el siguiente programa lee un número y luego escribe el número al cuadrado en la salida.

cuadrado.adb

```
1     -- Elevar un numero al cuadrado

3     with Ada.Text_IO;
4     use Ada.Text_IO;

6     procedure Cuadrado is

8          -- Necesario para leer/escribir Integers.
9          package Integer_InOut is new Integer_IO(Integer);
10         use Integer_InOut;

12         numero: Integer;
13    begin
14         Get(numero);
15         Put(numero ** 2);
16    end;
```

La sentencia *Get* lee desde teclado un entero (en este caso) y se lo asigna a la variable que pasamos como parámetro. A partir de la línea 14 del programa tendremos en la variable *numero* el número que haya escrito el usuario. Para utilizar el procedimiento *Get* hay que seguir las mismas normas que para utilizar el procedimiento *Put*. En particular, es preciso particularizar el paquete *Integer_IO* para el tipo de datos que queremos leer o escribir. Fíjate en que como ya dijimos estos procedimientos *Get* y *Put* no son los mismos que los utilizados para leer y escribir mensajes de texto (aunque se llamen igual).

5.5. Ordenar dos números cualesquiera

El siguiente programa lee dos números de la entrada estándar y los escribe ordenados en la salida. Para hacerlo, se utilizan dos variables: una para almacenar cada número. Tras llamar dos veces a *Get* para leer los números de la entrada (y almacenarlos en las variables), todo consiste en imprimirlos ordenados. Pero esto es fácil. Si el primer número es menor que el segundo entonces los escribimos en el orden en que los hemos leído. En otro caso basta escribirlos en el orden inverso.

ordenar2.adb

```
1     --
2     -- Escribir dos numeros ordenados
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure Ordenar2 is

10         package Integer_InOut is new Integer_IO(Integer);
11         use Integer_InOut;
```

```
13          a: Integer;
14          b: Integer;
15     begin
16          Get(a);
17          Get(b);
18          if a < b then
19               Put(a);
20               Put(b);
21          else
22               Put(b);
23               Put(a);
24          end if;
25     end;
—
```

5.6. Procedimientos

Una función no puede modificar los argumentos que se le pasan, pero es muy útil hacer subprogramas (como *Get*) que pueden modificar los argumentos o que pueden tener otros efectos laterales. Esos subprogramas se llaman **procedimientos** y en realidad ya los conocemos. En Ada, un programa es un procedimiento. Tanto *Get* como *Put* son procedimientos. El programa *Cuadrado* mostrado un poco más arriba también es un procedimiento que hemos utilizado como programa principal.

Hace poco hemos conocido la sentencia de asignación. Esta y otras sentencias que hemos visto antes corresponden a acciones que realiza nuestro programa. Pues bien,

un procedimiento es una acción con nombre

Por ejemplo, *Put* es en realidad un nombre para la acción o acciones que tenemos que realizar para escribir un valor. Igualmente, *Get* es en realidad un nombre para la acción o acciones que tenemos que realizar para leer una variable.

Veamos un ejemplo. Supongamos que tenemos el siguiente programa, que lee un número e imprime su cuadrado realizando esto mismo dos veces.

```
1  -- Elevar dos numeros al cuadrado

3  with Ada.Text_IO;
4  use Ada.Text_IO;

6  procedure Cuadrados is

8       -- Necesario para leer/escribir Integers.
9       package Integer_InOut is new Integer_IO(Integer);
10      use Integer_InOut;

12      numero: Integer;
13 begin
14      Get(numero);
15      Put(numero ** 2);
16      Get(numero);
17      Put(numero ** 2);
18 end;
```

Como podemos ver el cuerpo del programa realiza dos veces la misma secuencia:

```
    Get(numero);
    Put(numero ** 2);
```

Es mucho mejor inventarse un nombre para esta secuencia y hacer que el programa principal lo invoque dos veces. El resultado quedaría como sigue:

cuadrados.adb

```
1     -- Elevar dos numeros al cuadrado
2     with Ada.Text_IO;
3     use Ada.Text_IO;

5     procedure Cuadrados is

7           -- Necesario para leer/escribir Integers.
8           package Integer_InOut is new Integer_IO(Integer);
9           use Integer_InOut;

11          -- Escribir un numero leido al cuadrado
12          procedure Cuadrado is
13                numero: Integer;
14          begin
15                Get(numero);
16                Put(numero ** 2);
17          end;

19    begin
20          Cuadrado;
21          Cuadrado;
22    end;
```

Aquí podemos ver que hemos definido un procedimiento llamado *Cuadrado* de tal forma que el programa principal utiliza (dos veces) dicho procedimiento. Las líneas 20 y 21 invocan o llaman al procedimiento. Por tanto decimos, por ejemplo, que la línea 20 es una **llamada a procedimiento**. Como verás, la forma de definir un procedimiento es similar a la forma de definir una función, salvo por que se emplea la palabra reservada *procedure* en lugar de *function* y por que un procedimiento nunca devuelve ningún valor. Y si nos fijamos... ¡El programa principal es otro procedimiento más!

Si ejecutamos el programa sucederá lo de siempre: Ada comienza ejecutando *Cuadrados* justo en su *begin* en la línea 19. Cuando el flujo de control del programa alcanza la línea 20, Ada deja lo que estaba haciendo y llama al procedimiento *Cuadrado* (igual que sucede cuando llamamos a una función). A partir de este momento se ejecuta el procedimiento *Cuadrado*, hasta que éste alcance su ‘‘`end;`’’. En ese momento el procedimiento termina (decimos que **retorna**) y Ada continúa con lo que estaba haciendo (en este caso ejecutar el programa principal). En la línea 21 sucede lo mismo. El flujo de control llama al procedimiento *Cuadrado* y no continúa hasta que este retorne. La figura 5.3 muestra esto de forma esquemática.

Es bueno aprovechar ahora para ver algunas cosas respecto a la llamada a procedimiento:

- En la figura podemos ver cómo hay un único flujo de control (que va ejecutando una sentencia tras otra), representado por las flechas en la figura. Aun así, hay dos procedimientos en juego (tenemos *Cuadrados* y *Cuadrado*).
- Cuando se llama a *Cuadrado*, el procedimiento que hace la llamada deja de ejecutar hasta que el procedimiento llamado retorna.
- Podemos ver también que *Cuadrado* no empieza a existir hasta que alguien lo llama. Pasa lo mismo con el programa principal, al que llama el sistema operativo cuando le pedimos que ejecute el programa (no existe hasta que lo ejecutamos).

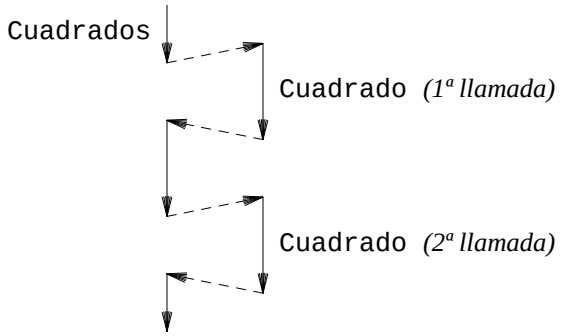


Figura 5.3: *Flujo de control de* `Cuadrados` *con dos llamadas a procedimiento*

- Podemos llamar al procedimiento más de una vez; y cada vez es una encarnación distinta (¿Será budista el sistema?). Por ejemplo, la variable *numero* declarada en *Cuadrado* no existe hasta la primera llamada y deja de existir cuando esta termina. Además, la variable *numero* que existe en la segunda llamada es *otra variable distinta* a la que teníamos en la primera llamada. Recuerda que las declaraciones de un procedimiento sólo tienen efecto hasta que el procedimiento termina.

Esto mismo pasa cuando *Cuadrado* llama a *Get* y cuando *Cuadrado* llama a *Put*. La figura 5.4 muestra esto. Las llamadas a procedimiento se pueden **anidar** sin problemas (un procedimiento puede llamar a otro y así sucesivamente).

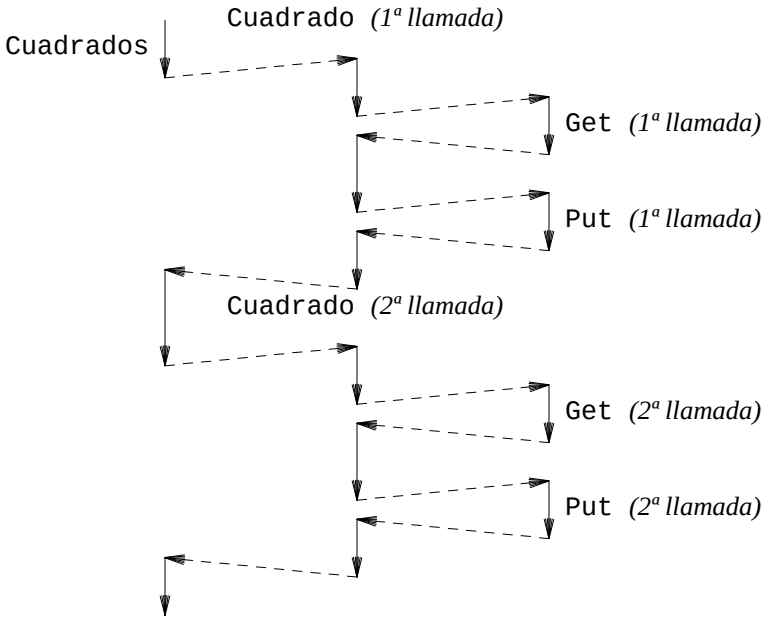


Figura 5.4: *Llamadas a procedimiento anidadas*

Intenta seguir el flujo de control del programa mirando el código y prestando atención a la figura 5.4. Debería resultarte cómodo trazar a mano el flujo de ejecución de un programa sobre el código. Sólo tienes que seguir el rastro de las llamadas y recordar a dónde tienes que retornar cuando cada llamada termina (El ordenador utiliza una estructura de datos llamada **pila** para hacer esto de forma automática).

5.7. Parámetros

Hemos visto cómo se producen las llamadas a procedimiento. Pero hemos omitido de la discusión el paso de parámetros.

Podemos declarar procedimientos con parámetros del mismo modo que declaramos parámetros en una función. Por ejemplo, el procedimiento *EscribeEntero* del siguiente programa escribe un entero tras un pequeño mensaje de tipo informativo y pasa a la siguiente línea tras escribir el entero. El programa principal lee dos números y escribe su suma.

suma.adb

```
1     -- suma dos numeros
2     with Ada.Text_IO;
3     use Ada.Text_IO;

5     procedure Suma is

7           -- Necesario para leer/escribir Integers.
8           package Integer_InOut is new Integer_IO(Integer);
9           use Integer_InOut;

11          procedure EscribeEntero(n: Integer) is
12          begin
13                Put("El valor es: ");
14                Put(n);
15                New_Line;
16          end;

18          numero1: Integer;
19          numero2: Integer;
20          resultado: Integer;
21    begin
22          Put("introduce dos numeros: ");
23          Get(numero1);
24          Get(numero2);
25          resultado := numero1 + numero2;
26          EscribeEntero(resultado);
27    end;
```

Las líneas 11 a 16 definen el procedimiento *EscribeEntero*. Como este subprograma escribe en la salida estándar, tiene efectos laterales y ha de ser un procedimiento. Esta vez el procedimiento tiene un parámetro declarado, llamado *n*, para el que hay que suministrar un argumento (como sucede en la llamada de la línea 26 en el programa principal).

Cuando se produce la llamada a *EscribeEntero* el procedimiento cobra vida. En este momento aparece una variable nueva: el parámetro *n*. El valor inicial de *n* es una copia del valor suministrado como argumento en la llamada (el valor de *resultado* en el programa principal en el momento de realizar la llamada en la línea 26).

Cuando el procedimiento llega a su ‘‘`end;`’’ el parámetro *n* (y cualquier variable que pueda tener declarada el procedimiento) deja de existir: su memoria deja de usarse y queda libre para otras necesidades futuras.

Una cosa curiosa es que podríamos haber llamado *resultado* al parámetro del procedimiento (usando el mismo nombre que el de una variable declarada en el programa principal). En tal caso, debería verse que nada varía con respecto a lo ya dicho: el parámetro del procedimiento es una variable distinta a la variable empleada en el programa principal. ¡Incluso si tienen el mismo nombre!

Cuando declaramos parámetros de este modo Ada copia el valor del argumento al parámetro, justo al principio de la llamada a subprograma. Además, Ada nos prohíbe modificar el valor del parámetro. Esto es, una sentencia que intente modificar *n*, dentro de *EscribeEntero*, dará un error en tiempo de compilación. Pero incluso si el lenguaje permite la modificación del parámetro, esta modificación es local al procedimiento y no sale del mismo.

La figura 5.5 muestra el proceso de suministrar valores para los parámetros de un subprograma. A este proceso se lo conoce como **paso de parámetros por valor**, dado que se pasa un valor (una copia) como parámetro. También se suele denominar **parámetros de entrada** a este tipo de parámetros, dado que sus valores entran del programa principal hacia el procedimiento pero no salen del procedimiento. Esto es, podemos utilizar este mecanismo para declarar e implementar un procedimiento como *Put*, pero no podemos utilizarlo para implementar un procedimiento como *Get.*

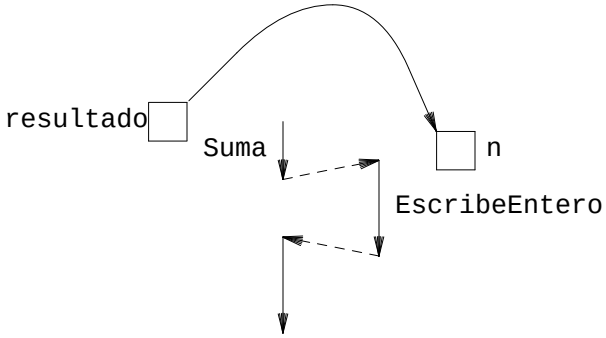


Figura 5.5: *Paso de parámetros por valor: se copia el valor del argumento al llamar.*

Pero veamos ahora un procedimiento que incrementa el valor de una variable. El siguiente programa imprime 2 en su salida estándar.

incr.adb

```
1     -- incrementa un numero
2     with Ada.Text_IO;
3     use Ada.Text_IO;

5     procedure Incr is

7           -- Necesario para leer/escribir Integers.
8           package Integer_InOut is new Integer_IO(Integer);
9           use Integer_InOut;

11          procedure Incrementar(n: in out Integer) is
12          begin
13                n := n + 1;
14          end;

16          numero: Integer;
17    begin
18          numero := 1;
19          Incrementar(numero);
20          Put(numero);
21          New_line;
22    end;
```

Comparando el procedimiento *Incrementar* con el procedimiento *EscribeEntero* programado antes podrá verse que la principal diferencia es que el parámetro está declarado utilizando las palabras reservadas *in out*.

Cuando un parámetro se declara como *in-out* dicho parámetro corresponde en realidad a la variable suministrada como argumento. Esto quiere decir que *el procedimiento es capaz de modificar el argumento*. Es como si el parámetro *n* fuese lo mismo que *numero* durante la llamada de la línea 19. La figura 5.6 muestra lo que sucede ahora durante la llamada.

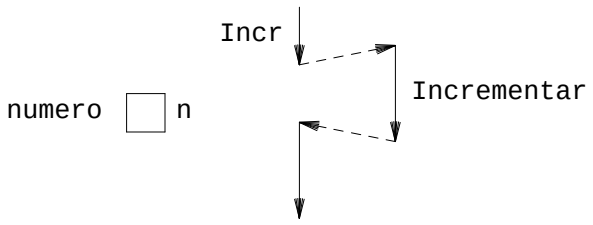


Figura 5.6: *Paso por referencia: durante la llamada el parámetro es lo mismo que el argumento*

Como puede verse no se produce copia alguna de parámetros. En su lugar, resulta que el nombre *n* se refiere también al valor que almacena la variable *numero* durante la llamada *Incrementar(numero).* Si el programa principal tuviese después una llamada de la forma *Incrementar(otronumero),* entonces *n* sería lo mismo que *otronumero* durante dicha llamada.

Vamos a verlo de otro modo. Dado que el parámetro *n* está declarado como *in out*, el procedimiento recibe una copia del valor que se ha utilizado como argumento (como sucedería por ejemplo si llamamos a una función) cuando se hace la llamada al procedimiento (esa es la parte *in*). Pero además, cuando termina el procedimiento, el valor de *n* se ve fuera del procedimiento, como si se copiase al argumento *numero* que se le pasó. Dicho de otro modo, dado que *n* es *in out, n* se refiere en realidad al argumento que ha recibido, esto es, a *numero*. Por tanto, la sentencia ‘‘`n := n + 1`’’ en el procedimiento está en realidad incrementando *numero*. A esta forma de pasar parámetros se la conoce como **paso por referencia**[1]. A la forma de paso de parámetros que hemos utilizado hasta ahora se la conoce como **paso por valor**. Recapitulemos:

1 El paso por referencia permite modificar el argumento que se pasa. El paso por valor no.

2 El paso por referencia requiere una variable como argumento. El paso por valor no. (Cualquier expresión del tipo adecuado basta).

3 El paso por valor suele implicar la necesidad de copiar el valor utilizado como argumento. El paso por referencia no suele implicarlo. (Esto es importante si una variable ocupa mucha memoria).

Podemos ver el paso de parámetros *in out* pensando en que se produce una copia del valor del argumento al parámetro a la entrada al procedimiento y se produce una copia del valor del parámetro al argumento a la salida del procedimiento (cuando este termina). Dicho de otra forma, el parámetro es de entrada (al procedimiento) y salida (del mismo). Pero en general, es bueno pensar que el paso de parámetros por referencia simplemente hace que los cambios al parámetro *se vean* tras el término del procedimiento; mientras que el paso de parámetros por valor hace que los cambios *sean locales* al procedimiento.

Por decir todo lo que hemos visto de otro modo: los parámetros que declaramos en las cabeceras de funciones y procedimientos son también variables. Eso si, son variables que están declaradas automáticamente al comenzar la ejecución del procedimiento del que son parámetro. Por tanto dejan de existir cuando el procedimiento (o función) termina.

---

[1] Técnicamente no es exactamente así, dado que el paso por referencia exige que se utilice una referencia al argumento y este paso de parámetros puede realizarse con dos copias. Nosotros vamos suponer que lo es. Hacer programas que sólo funcionen bien cuando el paso es en realidad por referencia da lugar a programas con fallos, por lo general. Lo importante es que el subprograma es capaz de modificar el argumento.

Cuando se llama a un procedimiento su parámetro (cada uno de los que tenga) se inicializa con una copia del argumento. Cuando se termina un procedimiento, si el parámetro está declarado como *in out* el último valor que tenga el parámetro (al final de la ejecución del procedimiento) se ve también en la variable que se ha suministrado como argumento. El parámetro deja de existir al terminar el procedimiento en cualquier caso.

Para no liarse, al principio, es conveniente dibujar en una hoja cajitas para las variables y parámetros y repasar mentalmente la ejecución del programa conforme se escribe, viendo qué valores se copian de qué variables a qué otras. Siguiendo mentalmente una traza de la ejecución del programa.

Notas adicionales sobre el paso de parámetros.

Ada permite una tercera forma de paso de parámetros. Esta en realidad es similar al paso por referencia salvo por que el compilador ignora el valor inicial que tienen los argumentos (Ada no realizará comprobaciones para ver si los valores encajan con el tipo de datos a que pertenecen). Hablamos de los **parámetros de salida**, o *out*. Para declararlos, utilizamos sólo la palabra reservada *out* en lugar de declararlos *in out*. Por ejemplo, este procedimiento que inicializa un entero declara su parámetro como *out*. En principio el compilador no garantiza que el valor del argumento esté disponible en el parámetro a la entrada del procedimiento, pero por lo demás el procedimiento es capaz de modificar el argumento como en el caso de parámetros por referencia.

```
1  procedure Inicializar(numero: out Integer) is
2  begin
3      numero := 0;
4  end;
```

En ocasiones hay que utilizar paso de parámetros por referencia en un procedimiento aún cuando no se desea modificar el parámetro. Cuando un parámetro tiene un tamaño considerable (arrays o matrices, por ejemplo) se suele utilizar paso de parámetros por referencia incluso si sólo se quiere consultar el parámetro. La razón es que en muchos lenguajes de programación el paso de parámetros por referencia evita hacer copia de los argumentos y en cambio el paso de parámetros por valor (en muchos lenguajes) supone hacer siempre una copia de los parámetros.

En este curso seguiremos esta costumbre, de tal forma que en general si un parámetro se considera que tiene un tamaño excesivo (algo más de algunos bytes) se empleará *in out* en su declaración, aunque sólo se desee consular éste. Nótese que esto puede hacer que ciertas funciones deban ser procedimientos, dado que una función nunca puede utilizar parámetros por referencia.

5.8. Variables globales

A las variables definidas fuera de un procedimiento se las denomina **variables globales** o **externas**. En esta asignatura **no se permite el uso variables externas a un subprograma** (en un procedimiento o función). Esto es, incluso cuando el lenguaje lo permita, en este curso no se puede desde un procedimiento o función utilizar nada que no forme parte de la cabecera del procedimiento. Naturalmente si que podemos utilizar constantes, funciones y procedimientos y todo tipo de expresiones y estructuras de control (decisiones, etc.). Pero el diálogo entre el procedimiento y el resto del programa debe quedar restringido a la cabecera del procedimiento. Lo mismo queda dicho para las funciones.

Pensemos que si en el programa anterior el procedimiento *Incrementar* intentase utilizar directamente *numero* entonces ese procedimiento sólo serviría para incrementar esa variable. ¡Ninguna otra!. Además, en programas mas largos sería fácil olvidar que un procedimiento está haciendo cosas más allá de lo que le permite su cabecera. De nuevo, eso hace que el programa tenga errores a no ser que seamos capaces de tener *todos* los detalles del programa en la cabeza. Y no lo somos.

Lo bueno de utilizar subprogramas y de hacer que un subprograma sólo pueda hablar con el resto del programa empleando su cabecera es que podemos **olvidarnos** por completo del resto del programa mientras programamos el subprograma. Como ya se ha visto al utilizar funciones, es muy útil emplear la idea del **refinamiento progresivo** y hacer el programa **top-down** (de arriba a abajo). Haciendo programas que llaman a otros más sencillos, que a su vez llaman a otros más sencillos, etc.

Programar consiste en inventar subprogramas que resuelven subproblemas del que tenemos entre manos, olvidándonos de cómo se hacen esos subproblemas inicialmente. Y luego, aplicar la misma técnica a los subproblemas hasta resolverlos. Aunque todavía no lo hemos dicho, también se hace lo mismo con los datos a la hora de organizarlos.

Dadas las normas respecto al orden de las declaraciones que utilizamos en este curso (las variables han de seguir a los subprogramas) y dado que no permitimos anidar declaraciones de subprogramas (aunque Ada lo permite) no es posible utilizar variables globales por accidente salvo violando dichas normas.

5.9. Visibilidad y ámbito

Consideremos el siguiente programa.

ambitos.adb

```
1     -- Programa absurdo que ademas viola las normas
2     -- para ayudar con la comprension del uso de variables
3     procedure Ambitos is

5           n: Integer;     -- (1)

7           procedure Proc1(n: Integer; n2: in out Integer) is
8                n: Integer;
9           begin
10               n2 := n ** 2;
11          end;

13          procedure Proc2(n: Integer) is
14          begin
15               Proc1(3, n);
16               Put(n);
17          end;

19    begin
20          Get(n);
21          Proc2(n);
22    end;
```

Hay muchas variables y parámetros que se llaman *n*. Pero todas son distintas. Veamos. La variable que tiene un comentario marcándola como ‘‘`(1)`’’ existe durante la ejecución de todo el programa. Empieza a existir cuando se crean las variables del programa principal (justo antes de empezar a ejecutar el cuerpo del programa principal). Y sigue existiendo hasta que el flujo de control alcanza el *end* del programa principal, luego existe durante todo el programa. La llamada a *Get* lee la variable de la que hablamos de la entrada.

Cuando llamamos a *Proc2* aparece una **nueva** variable: el parámetro *n*. Esa nueva variable, también llamada *n*, es distinta de la variable etiquetada como ‘‘`(1)`’’ en el programa. En la llamada a *Proc2* la *n* del parámetro se inicializa con una copia del valor que hay en la *n* del programa principal. Este parámetro comienza a existir al llamar a *Proc2* y deja de existir cuando *Proc2* llegue a su *end*.

Ahora *Proc2* llama a *Proc1* utilizando 3 como primer argumento y *n* (el parámetro *n* de *Proc2*) como segundo argumento. Esto quiere decir que empezamos a ejecutar *Proc1*. Ahora aparecen otras dos nuevas variables: *n* y *n2*. Aunque esta nueva *n* se llama igual que las dos anteriores, es otra distinta. Esta última *n* se inicializa en nuestra llamada a *Proc1* con una copia del valor 3 que se usó como primer argumento. Igualmente, *n2* se inicializa en la llamada con una copia del valor del segundo argumento, que es el valor de la variable *n* en *Proc2*. Esto es, el valor del parámetro *n* de *Proc2*.

Tras la llamada a *Proc1*, pero antes de empezar a ejecutar su cuerpo, las cosas empeoran. Este procedimiento declara una variable llamada *n*. Esta variable **oculta** el parámetro del mismo nombre, dado que no puede haber dos variables en un único bloque de sentencias que se llamen igual. De hecho, esta declaración no se puede hacer. Si la hacemos (si el lenguaje nos deja) estamos tapando un parámetro y perdiéndolo, con lo que *Proc1* no va a hacer lo que debería.

Cuando *Proc1* termina su ejecución sus variables y sus parámetros dejan de existir. Cuando luego *Proc2* termina, sucede igual con los parámetros de *Proc2*.

Otra nota curiosa. Desde *Proc2* podemos llamar a *Proc1*, pero no al contrario. Esto es así puesto que el código de *Proc2* tiene declarado anteriormente el procedimiento *Proc1*, con lo que este procedimiento existe para el y se le puede llamar. En el punto del programa en que *Proc1* está definido no existe todavía ningún procedimiento *Proc2*, por tanto no lo podemos llamar.

Se llama **ámbito** de una variable (o de un objeto en general) al trozo del programa en que dicha variable existe. Las variables declaradas en un procedimiento tienen como ámbito el trozo del programa que comprende desde la declaración hasta el fin de ese procedimiento. Se dice que son **variables locales** de ese procedimiento. Cada llamada al procedimiento crea una copia nueva de dichas variables. Cada retorno (o fin) del procedimiento las destruye. Las variables externas se dice que son **variables globales**.

Otro concepto importante es el de **visibilidad**. Una variable es visible en un punto del programa si ese punto está dentro del ámbito de la variable y si además no hay ninguna otra variable que tenga la desfachatez de usar el mismo nombre y ocultarla. Por ejemplo, el parámetro *n* del procedimiento *Proc1* está dentro de su ámbito en el cuerpo de ese procedimiento, pero no es visible dado que la variable local lo oculta. La variable marcada como ‘‘`(1)`’’ tiene como ámbito el programa entero (es una variable global). No obstante, esta variable sólo es visible en el cuerpo del programa principal dado que todos los subprogramas declaran variables y/o argumentos con el mismo nombre, ocultándola.

Recordamos de nuevo que, lo permita el lenguaje o no, en este curso no se permite declarar variables que oculten parámetros ni tampoco declarar variables globales.

5.10. Ordenar puntos

Queremos ordenar dos puntos situados sobre una recta discreta, tras leerlos de la entrada estándar, según su distancia al origen. El programa debe escribir luego las distancias de menor a mayor. Por ejemplo, si tenemos los puntos mostrados en la figura 5.7, el programa debería escribir en su salida estándar:

```
2
3
```

Vamos a resolver este problema empleando las nuevas herramientas que hemos visto en este capítulo.

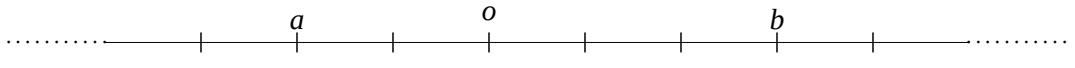


Figura 5.7: *Queremos ordenar dos puntos a y b según su distancia a o.*

Sabemos que tenemos definido un punto con un valor entero. El problema consiste en

1 leer dos puntos

2 ordenarlos de menor a mayor distancia

3 escribir las distancias de ambos.

Luego, suponiendo que tenemos disponibles cuantos procedimientos y funciones sean necesarios, podríamos escribir el programa como sigue:

```
1  --
2  -- Ordenar puntos segun sus distancias al origen y escribir las distancias.
3  --

5  with Ada.Text_IO;
6  use Ada.Text_IO;

8  procedure Ordenar is

10     ...

12     a: Integer;
13     b: Integer;
14 begin
15     LeerPunto(a);
16     LeerPunto(b);
17     Ordenar(a, b);
18     EscribirDistancia(a);
19     EscribirDistancia(b);
20 end;
```

Hemos declarado dos variables, *a* y *b*, para almacenar los puntos. Después llamamos a un procedimiento *LeerPunto* para leer el punto *a* y de nuevo para leer el punto *b*.

Como puede verse, para cada objeto que queremos que manipule el programa declaramos una variable (de tipo *Integer* puesto que para nosotros un punto es un valor entero).

Igualmente, nos hemos inventado un procedimiento *LeerPunto*, puesto que necesitamos leer un punto de la entrada. Debe ser un procedimiento y no una función puesto que va la leer de la entrada, lo que es un efecto lateral. Al procedimiento hemos de pasarle una variable (para un punto) que debe quedar inicializada con el valor leído de la entrada. Ya veremos cómo lo hacemos.

Una vez tengamos leídos los dos puntos querremos ordenarlos (según sus distancias). De nuevo, suponemos que tenemos ya programado un procedimiento *Ordenar*, capaz de ordenar dos puntos. Para que los ordene, le tendremos que suministrar los dos puntos. Y supondremos que, tras la llamada a *Ordenar*, el primer argumento, *a*, tendrá el punto con menor distancia y el segundo argumento, *b*, tendrá el punto con mayor distancia. Como *a* y *b* son variables y no son constantes, no hay ningún problema en que cambien de valor. Para nosotros *a* y *b* son los dos puntos de nuestro programa, pero nos da un poco igual cuál es *a* y cuál es *b*.

Nos resta escribir las distancias en orden, para lo que (de nuevo) nos inventamos un procedimiento *EscribirDistancia* que se ocupe de escribir la distancia para un punto.

Un detalle importante es que aunque nos hemos inventado estos subprogramas conforme los necesitamos, hemos intentado que sean siempre subprogramas útiles en general y no subprogramas que sólo hagan justo lo que debe hacer este programa. Por ejemplo, no hemos supuesto que tenemos un procedimiento *EscribirDistancias(a*,b) que escribe dos distancias. Sería raro que lo tuviéramos. Pero sí podría ser razonable que alguien hubiese programado antes un *EscribirDistancia(a)* y lo pudiéramos utilizar ahora, por eso hemos supuesto directamente que ya lo tenemos.

Para leer un punto podríamos escribir un mensaje (para que el usuario sepa que el programa va a detenerse hasta que se escriba un número en la entrada y se pulse un *Enter*) y luego leer un entero. Ahora bien, el procedimiento *LeerPunto* debe ser capaz de modificar su argumento. Dicho de otro modo, su parámetro debe de ser de entrada/salida o debe pasarse por referencia. Por lo tanto, podemos programar:

```
1  procedure LeerPunto(punto: in out Integer) is
2  begin
3      Put("Introduce la posicion x del punto: ");
4      Get(punto);
5  end;
```

El procedimiento *Get* estará declarado de un modo similar, por eso es capaz de modificar *punto* en la llamada de la línea 4. Por cierto, podríamos haber declarado *punto* sólo como *out*.

El siguiente problema es cómo ordenar los puntos. Aquí tenemos dos problemas. Por un lado, el procedimiento debe ser capaz de modificar ambos argumentos (luego ambos deben pasarse por referencia). Por otro lado, tenemos que ver cómo los ordenamos.

Para ordenarlos, supondremos de nuevo que tenemos una función *DistanciaLineal* que nos devuelve la distancia de un punto a un origen. Suponiendo esto, podemos ver si la distancia del primer punto es menor que la distancia del segundo punto. En tal caso ya los tendríamos ordenados y no tendríamos que hacer nada. En otro caso tendríamos que intercambiar ambos puntos para dejarlos ordenados. Pues bien, programamos justo esto:

```
1  -- ordenar segun la distancia al origen de menor a mayor.
2  procedure Ordenar(a: in out Integer; b: in out Integer) is
3  begin
4      if DistanciaLineal(Cero, a) > DistanciaLineal(Cero, b) then
5             Intercambiar(a, b);
6      end if;
7  end;
```

Ambos parámetros están declarados *in-out*, puesto que queremos modificarlos. Nos hemos inventado la constante *Cero* para representar el origen, la función *DistanciaLineal*, y el procedimiento *Intercambiar*. Este último ha de ser un procedimiento puesto que debe modificar sus argumentos y además no devuelve ningún valor.

La función *DistanciaLineal* ya la teníamos programada antes, por lo que sencillamente la reutilizaremos en este programa (copiando su código).

El procedimiento *Intercambiar* requiere más trabajo. Desde luego, va a recibir dos parámetros *in-out*, digamos:

```
procedure Intercambiar(a: in out Integer; b: in out Integer)
```

Pero pensemos ahora en su cuerpo. Podríamos utilizar esto:

```
a := b;
b := a;
```

Ahora bien, digamos que *a* vale 3 y *b* vale 4. Si ejecutamos

```
a := b;
```

dejamos en *a* el valor que tiene *b*. Luego ambas variables pasan a valer 4. Si ahora ejecutamos

```
b := a;
```

entonces dejamos en *b* el valor que *ahora* tenemos en *a*. Este valor era el valor que teníamos en *b*, 4. Luego hemos hecho que ambas variables tengan lo que hay en *b* y hemos perdido el valor que había en *a*.

La forma de arreglar este problema y conseguir intercambiar los valores es utilizar una **variable auxiliar**. Esta variable la vamos a utilizar sólo para mantener temporalmente el valor de *a*, para no perderlo y actualizar *b* después. El procedimiento quedaría como sigue

```
1  procedure Intercambiar(a: in out Integer; b: in out Integer) is
2      aux: Integer;
3  begin
4      aux := a;
5      a := b;
6      b := aux;
7  end;
```

Como vemos, el interfaz o cabecera del procedimiento sigue siendo el mismo: recibe dos puntos y los ordena, dejándolos ordenados a la salida. Pero declaramos una variable *aux* para mantener uno de los valores de forma temporal y en lugar de actualizar *b* a partir de *a* lo actualizamos a partir del valor que teníamos inicialmente en *a*, esto es, a partir de *aux*.

Nos falta implementar *EscribirDistancia*. Este subprograma debe ser un procedimiento dado que corresponde a algo que tenemos que hacer y no a un valor que tenemos que calcular. Por no mencionar que tiene efectos laterales. Dado que no precisa modificar su parámetro este ha de pasarse por valor y no por referencia. Como el procedimiento necesitará calcular la distancia para el punto podemos utilizar la función *DistanciaLineal* que ya teníamos una vez mas. El programa completo nos ha resultado como sigue.

ordenarpuntos.adb

```
1     --
2     -- Ordenar puntos segun sus distancias al origen y escribir las distancias.
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure OrdenarPuntos is

10          -- Origen en nuestra linea discreta
11          Cero: constant Integer := 0;

13          package Integer_InOut is new Integer_IO(Integer);
14          use Integer_InOut;

16          -- distancia entre dos puntos en una recta discreta
17          function DistanciaLineal(a: Integer; b: Integer) return Integer is
18          begin
19               if a > b then
20                    return a -b;
21               else
22                    return b-a;
23               end if;
24          end;

26          procedure LeerPunto(punto: in out Integer) is
27          begin
28               Put("Introduce la posicion x del punto: ");
29               Get(punto);
30          end;
```

```
32          procedure EscribirDistancia(punto: Integer) is
33          begin
34               Put("punto :");
35               Put(punto);
36               Put(" distancia: ");
37               Put(DistanciaLineal(Cero, punto));
38               New_Line;
39          end;

41          procedure Intercambiar(a: in out Integer; b: in out Integer) is
42               aux: Integer;
43          begin
44               aux := a;
45               a := b;
46               b := aux;
47          end;

49          -- ordenar segun la distancia al origen de menor a mayor.
50          procedure Ordenar(a: in out Integer; b: in out Integer) is
51          begin
52               if DistanciaLineal(Cero, a) > DistanciaLineal(Cero, b) then
53                    Intercambiar(a, b);
54               end if;
55          end;

57          a: Integer;
58          b: Integer;
59     begin
60          LeerPunto(a);
61          LeerPunto(b);
62          Ordenar(a, b);
63          EscribirDistancia(a);
64          EscribirDistancia(b);
65     end;
—
```

5.11. Resolver una ecuación de segundo grado

Queremos resolver cualquier ecuación de segundo grado. Recordamos que una ecuación de segundo grado adopta la forma

$$ax^2 + bx + c = 0$$

y que, de tenerlas, la ecuación tiene en principio dos soluciones con fórmulas

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad y \quad \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

De nuevo, procedemos como de costumbre. Lo primero es ver cómo representar la ecuación y después nos inventamos cuantos subprogramas necesitemos. La primera tarea es fácil: para nosotros la ecuación son tres coeficientes: *a*, *b* y *c*; todos ellos números reales.

Luego nuestro programa debe leer la ecuación, ver si tiene solución, calcular la solución e imprimir la solución. Es justo esto lo que programamos por ahora.

```
1  --
2  -- Resolver una ecuacion de segundo grado.
3  --
```

```
5  with Ada.Text_IO;
6  use Ada.Text_IO;

8  procedure Ec2grado is

10     package Float_InOut is new Float_IO(Float);
11     use Float_InOut;

13     ...

15     -- coeficientes de la ecuacion.
16     a: Float;
17     b: Float;
18     c: Float;
19     -- soluciones de la ecuacion.
20     sol1: Float;
21     sol2: Float;
22 begin
23     LeerEcuacion(a, b, c);
24     if ExisteSolucionReal(a, b, c) then
25            CalcularSolucion(a,b, c, sol1, sol2);
26            ImprimirSolucion(sol1, sol2);
27     else
28            Put("No hay solucion unica real");
29            New_Line;
30     end if;
31 end;
```

Para manipular la ecuación estamos utilizando tres variables de tipo *Float*, para almacenar los coeficientes. Dado que a estos coeficientes se los conoce como *a*, *b* y *c*, esos han sido los nombres para las variables. Por otro lado sabemos que tendremos (generalmente) dos soluciones, que tendremos que manipular. Para ello declaramos otras dos variables más, llamadas *sol1* y *sol2*.

Siguiendo nuestra costumbre, nos hemos inventado sobre la marcha los subprogramas *LeerEcuacion*, para leer la ecuación, *ExisteSolucionReal*, que nos dirá cuándo existe solución (real, no compleja) a la ecuación, *CalcularSolucion*, que calcula dicha solución (recordemos que estará compuesta de dos posibles valores para *x* en realidad) e *ImprimirSolucion*, para que informe de la solución.

No importa si inicialmente hubiésemos programado algo como

```
LeerEcuacion(a, b, c);
CalcularSolucion(a,b, c, sol);
ImprimirSolucion(sol);
```

En cuanto programemos unas versiones, inicialmente vacías, de estos procedimientos y empecemos a utilizar el programa nos daremos cuenta de dos cosas: (1) no siempre tenemos solución y (2) la solución son dos valores diferentes en la mayoría de los casos. En este punto cambiamos el programa para tener esto en cuenta y a otra cosa.

Leer la ecuación requiere tres parámetros, por referencia los tres. Los tres son en realidad parámetros de salida (el valor inicial de los argumentos da igual).

Para ver si existe una solución necesitamos que *a* sea distinto de cero, puesto que de otro modo tendremos problemas al hacer la división por $2a$. Además, si queremos sólo soluciones reales necesitamos que b^2 sea mayor o igual que $4ac$.

El cálculo de la solución requiere tres parámetros cuyo valor necesitamos (los pasamos por valor) para definir la ecuación y dos parámetros que vamos a devolver (los pasamos por referencia como parámetros de salida).

Por último, imprimir la solución no requiere modificar los parámetros. Podríamos habernos molestado menos en cómo escribimos la solución, pero parecía adecuado escribir un único valor si ambas soluciones son iguales. Nótese que el código común en ambas ramas del *if* en esta función se ha extraído de dicha sentencia, para no repetirlo.

El programa queda como sigue:

ec2grado.adb

```
1     --
2     -- Resolver una ecuacion de segundo grado.
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;
7     with Ada.Numerics.Generic_Elementary_Functions;

9     procedure Ec2grado is

11          package Float_InOut is new Float_IO(Float);
12          use Float_InOut;

14          package Math is new Ada.Numerics.Generic_Elementary_Functions(Float);
15          use Math;

17          procedure LeerEcuacion(a: out Float; b: out Float; c: out Float) is
18          begin
19               Put("Escribe los coeficientes a, b y c: ");
20               Get(a);
21               Get(b);
22               Get(c);
23          end;

25          function ExisteSolucionReal(a: Float; b: Float; c: Float) return Boolean is
26          begin
27               return a /= 0.0 and b ** 2 >= - (4.0 * a * c);
28          end;

30          procedure CalcularSolucion(a: Float; b: Float; c: Float;
31                                     sol1: out Float; sol2: out Float) is
32               discriminante: Float;
33          begin
34               discriminante := b ** 2 - (4.0 * a * c);
35               sol1 := ((-b) + sqrt(discriminante)) / (2.0 * a);
36               sol2 := ((-b) - sqrt(discriminante)) / (2.0 * a);
37          end;
```

```
39         procedure ImprimirSolucion(sol1: Float; sol2: Float) is
40         begin
41              Put("Solucion:");
42              if sol1 = sol2 then
43                   Put("x=");
44                   Put(sol1);
45              else
46                   Put("x1=");
47                   Put(sol1);
48                   Put("x2=");
49                   Put(sol2);
50              end if;
51              New_Line;
52         end;
53
54         -- coeficientes de la ecuacion.
55         a: Float;
56         b: Float;
57         c: Float;
58         -- soluciones de la ecuacion.
59         sol1: Float;
60         sol2: Float;
61    begin
62         LeerEcuacion(a, b, c);
63         if ExisteSolucionReal(a, b, c) then
64              CalcularSolucion(a,b, c, sol1, sol2);
65              ImprimirSolucion(sol1, sol2);
66         else
67              Put("No hay solucion unica real");
68              New_Line;
69         end if;
70    end;
—
```

Este programa en realidad tiene un error, la comparación

```
sol1 = sol2
```

está comparando dos números reales. En general no sabemos si son iguales por que la precisión de los números reales en el ordenador no da para más o si realmente son iguales. Tampoco sabemos en general si los dos números son distintos por que los cálculos con números aproximados tienen resultados aproximados y, claro, podríamos terminar con dos números que aunque deben ser iguales no lo son realmente. No obstante, en este caso, lo que nos interesa es no imprimir dos veces justo el mismo valor, por lo que parece que esta comparación basta.

Como podrás ver los programas realmente no se terminan nunca y siempre pueden mejorarse. Por ejemplo, podríamos hacer que el programa supiese manipular soluciones complejas. No obstante hay que saber cuando decir basta.

Problemas

Cuando el enunciado corresponda a un problema ya hecho te sugerimos que lo vuelvas a hacer sin mirar la solución.

1 Leer un número de la entrada estándar y escribir su tabla de multiplicar.

2 Hacer esto mismo sin emplear ningún literal numérico salvo el ‘‘`1`’’ en el programa. Sugerencia: utilizar una variable para mantener el valor del factor que varía en la tabla de multiplicar y ajustarla a lo largo del programa.

3 Leer dos números de la entrada estándar y escribir el mayor de ellos.

4 Intercambiar dos variables de tipo entero.

5 Leer tres números de la entrada estándar y escribirlos ordenados.

6 Suponiendo que para jugar a los barcos queremos leer una casilla del tablero (esto es, un carácter de la A a la K y un número del 1 al 10). Haz un programa que lea una casilla del tablero y que imprima las casillas que la rodean. Supongamos que la casilla que leemos no está en el borde del tablero.

7 Haz un programa que lea una casilla del juego de los barcos y diga si la casilla está en el borde del tablero o no.

8 Haz que el programa 4 funcione correctamente si la casilla está en el borde del tablero.

9 Haz un programa que lea un dígito hexadecimal y lo convierta a decimal.

10 Haz el mismo programa pero para números de cuatro dígitos.

11 Haz un programa que lea la coordenada del centro de un círculo y valor para el radio del círculo y luego la coordenada de un punto y determine si el punto está dentro del círculo.

12 Haz un programa que escriba *ADA* en letras grandes empleando ‘‘*’’ como carácter para dibujar las letras. Cada letra ha de escribirse bajo la letra anterior, como en un cartel de anuncio vertical.

13 Haz un programa que escriba *DADADADADADA* con las mismas directrices del problema anterior.

14 Cambia el programa anterior para que el usuario pueda indicar por la entrada qué carácter hay que utilizar para dibujar la letra *A* y qué carácter hay que utilizar para dibujar la letra *D*.

15 Si no lo has hecho, utiliza procedimientos y funciones de forma intensiva para resolver todos los ejercicios anteriores de este curso.

16 Haz que los ejercicios anteriores de este curso que realizan cálculos para valores dados sean capaces de leer dichos valores de la entrada.

6 — Tipos escalares y tuplas

6.1. Otros mundos

En los programas que hemos realizado hemos utilizado **tipos de datos primitivos** de Ada. Esto es, tipos que forman parte del mundo según lo ve Ada: *Integer*, *Float*, *Character* y *Boolean*.

Pero hemos tenido problemas de claridad y de falta de abstracción a la hora de escribir programas que manipulen entidades que no son ni enteros ni reales ni caracteres ni valores de verdad. Por ejemplo, si queremos manipular fechas vamos a necesitar manipular meses y también días de la semana. Si hacemos un programa para jugar a las cartas vamos a tener que entender lo que es Sota, Caballo y Rey. Hasta el momento hemos tenido que utilizar enteros para ello. En realidad nos da igual si el ordenador utiliza siempre enteros para estas cosas. Lo que no queremos es tenerlo que hacer nosotros.

Utilizar enteros y otros tipos conocidos por Ada para implementar programas que manipulan entidades abstractas (que no existen en el lenguaje) es un error. Los programas resultarán complicados rápidamente y pronto no sabremos lo que estamos haciendo. Cuando veamos algo como

```
if c = 8 then
       ...
end if
```

no sabremos si *c* es una carta y ‘‘`8`’’ es en realidad una *Sota* o qué estamos haciendo (a esto ayuda el críptico nombre ‘‘`c`’’ elegido para este ejemplo). Tendremos problemas serios para entender nuestro propio código y nos pasaremos mucho tiempo localizando errores y depurándolos.

Ada permite **definir nuevos tipos** para inventarnos mundos que no existen inicialmente en Ada (ver figura 6.1). Como ya sabemos un tipo de datos no es otra cosa que un conjunto de elementos identificado por un nombre, con una serie de operaciones y al que pertenecen una serie de entidades homogéneas entre sí.

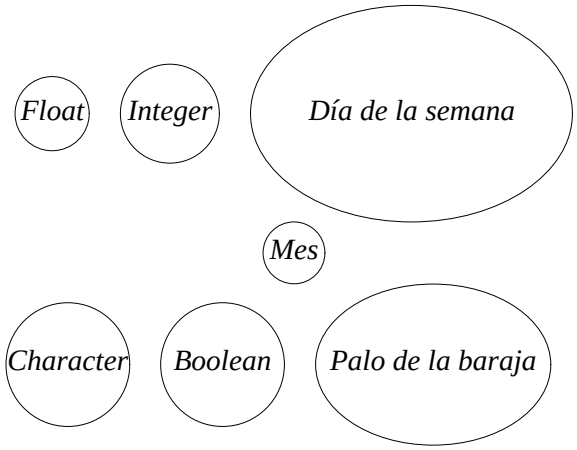


Figura 6.1: *Además de enteros, caracteres, reales y booleanos podemos inventar nuevos mundos.*

Pues bien, una forma de inventarse un nuevo mundo (al que pertenecerán entidades que no pueden mezclarse con las de otros mundos) es declarar un nuevo tipo de datos. Una de las formas que tenemos para hacer esto es **enumerar** los elementos del nuevo tipo de datos. A los tipos definidos así se los conoce como **tipos enumerados**. Por lo tanto, en un tipo enumerado los

elementos (los literales) son enumerables y se puede asignar un número natural a uno.

Esta declaración define un nuevo tipo enumerado llamado *TipoDiaSem*, para representar los días de la semana:

```
type TipoDiaSem is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
```

Esta otra declaración define un nuevo tipo enumerado para los meses del año:

```
type TipoMes is (Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov, Dic);
```

Esta otra podría utilizarse para el valor de una carta:

```
type TipoValorCarta is (Uno, Dos, Tres, Cuatro, Cinco,
                    Seis, Siete, Sota, Caballo, Rey);
```

Naturalmente podríamos también definir:

```
type TipoPaloBaraja is (Bastos, Oros, Copas, Espadas);
```

O tal vez queramos manipular dígitos de números romanos:

```
type TipoDigitoRomano is (I, V, X, L, C, D, M);
```

Como puede verse, inventarse otro tipo es fácil para tipos enumerados. Basta enumerar con esta sintaxis los elementos del conjunto. Tras cada una de estas declaraciones disponemos de un nuevo tipo de datos. Por ejemplo, tras la primera declaración mostrada disponemos del nuevo identificador *TipoDiaSem* que corresponde con el tipo de datos para días de la semana; tras la segunda, *TipoMes*; etc.

Todos los nombres que hemos escrito entre paréntesis a la hora de enumerar los elementos del nuevo tipo son, a partir del momento de la declaración del tipo, nuevos literales que podemos utilizar en el lenguaje. Naturalmente, son literales del tipo en el que los hemos enumerado. Así por ejemplo, *Lun* y *Vie* son a partir de ahora literales (valores) del tipo *TipoDiaSem*.

Para el lenguaje, *Vie* no es muy diferente de ‘‘3’’. La diferencia radica en que el primero es de tipo *TipoDiaSem* y el segundo de tipo *Integer*. De hecho, *Vie* estará representado dentro del ordenador por un número entero, por ejemplo el 4 o el 5.

Esto quiere decir que ahora podríamos por ejemplo declarar una variable del tipo `TipoDiaSem` como sigue:

```
dia: TipoDiaSem;
```

O una constante, como hacemos a continuación:

```
UnBuenDia: constant TipoDiaSem := Vie;
```

Si es que consideramos el viernes como una constante universal para un buen día.

Es importante utilizar siempre el mismo estilo para los identificadores de nuevos tipos. Principalmente, resulta útil distinguirlos de variables, funciones y otros artefactos tan sólo con ver sus nombres. En este curso los nombres de tipo deberían siempre comenzar por ‘‘`Tipo`’’ y capitalizarse de forma similar a como lo hacemos para funciones y procedimientos. Los identificadores para los elementos del tipo también estarán capitalizados del mismo modo, con la inicial en mayúscula. Queremos nombres fáciles de escribir y, sobre todo, fáciles de leer. Estos nombres deberían ser nombre cortos y claros, pero eso sí, es bueno que no sean nombres populares para nombrar variables o funciones. (No deberían coincidir con nombres de procedimiento, que deberían corresponder a las acciones que realizan). Por ejemplo, los literales empleados para definir un dígito romano son pésimos. Posiblemente fuese mejor utilizar algo como

```
type TipoDigitoRomano is (IRomano, VRomano, XRomano,
                    LRomano, CRomano, DRomano, MRomano);
```

¡En realidad ya conocíamos los tipos enumerados! Los enteros, los caracteres y los booleanos son tipos enumerados (podemos contarlos y sus literales están definidos por un único valor). La principal diferencia entre estos tipos y los nuevos es que no tenemos que definirlos (están predefinidos en el lenguaje). Así pues, debería resultar sencillo manipular valores y variables de los nuevos tipos enumerados al escribir nuestro programa: lo haremos exactamente igual que hemos hecho con cualquier tipo enumerado de los predefinidos o primitivos del lenguaje.

Por ejemplo, sabemos que este programa lee un carácter, calcula el siguiente carácter y lo imprime (Suponiendo que el carácter leído no es el último):

```
1  procedure SiguienteChar is

3      c: Character;
4  begin
5      Get(c);
6      c := Character'Succ(c);
7      Put(c);
8  end;
```

Pues bien, el siguiente programa lee un día de la semana, calcula el siguiente día y lo imprime (aunque justo este programa no compilará por lo que veremos luego). Todo esto también suponiendo que el día que ha leído no es el último:

```
1  procedure SiguienteDia is

3      type TipoDiaSem is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);

5      dia: TipoDiaSem;
6  begin
7      Get(dia);
8      dia := TipoDiaSem'Succ(dia);
9      Put(dia);
10 end;
```

Igual que deberíamos haber hecho con el programa para el siguiente carácter, haríamos bien en no intentar calcular el siguiente día al domingo: esto es un error y el programa se detendría. Podemos por ejemplo considerar que a un domingo le sigue un lunes y cambiar el programa según se ve a continuación:

```
1  procedure SiguienteDia is

3      type TipoDiaSem is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);

5      dia: TipoDiaSem;
6  begin
7      Get(dia);
8      if dia = Dom them
9             dia := Lun;
10     else
11            dia := TipoDiaSem'Succ(dia);
12     end if;
13     Put(dia);
14 end;
```

Como puede verse, podemos declarar, inicializar, asignar y consultar variables de un tipo enumerado del mismo modo que si fuesen variables de cualquier otro tipo. Además, la comparación de igualdad también está definida y podemos comparar variables y literales. Los atributos como *Succ* también están definidos. En general, todo lo que podemos hacer con *Character* también lo podemos hacer con un tipo enumerado (Los enteros y los booleanos son un poco especiales puesto que tienen operaciones exclusivas de ellos, como las aritméticas para los

enteros y las del álgebra de bool para los booleanos).

Por ejemplo, el *if* del programa anterior que trata de evitar calcular el día siguiente al domingo (puesto que no existen más días) podríamos haberlo escrito como sigue.

```
8       if dia = TipoDiaSem'Last them
9               dia := TipoDiaSem'First;
10      else
11              dia := TipoDiaSem'Succ(dia);
12      end if;
```

Otro detalle importante es que podemos leer días de la entrada estándar y escribirlos en la salida estándar. Igual que particularizamos paquetes para hacer entrada/salida de ciertos tipos, podemos también hacerlo para hacer entrada/salida de cualquier tipo enumerado:

```
-- particulariza Enumeration_IO para el tipo TipoDiaSem
package DiaSem_IO is new Enumeration_IO(TipoDiaSem);
use DiaSem_IO;
```

Y luego es cuestión de usar

```
Put(dia);
```

y

```
Get(dia);
```

¡Ya sabemos por qué no compilaba el programa que mostramos antes!

6.2. Submundos: subconjuntos de otros mundos

Hay otra forma de declarar nuevos conjuntos de elementos. Por ejemplo, en la realidad (en matemáticas) tenemos los enteros pero también tenemos los números positivos. Igualmente, tenemos los días de la semana pero también tenemos los días laborables. Los positivos son un subconjunto de los enteros y los días laborables son un subconjunto de los días de la semana. En Ada tenemos **subtipos** para expresar este concepto. Un subtipo es a un tipo lo que un subconjunto es a un conjunto.

Un subtipo es un nuevo nombre para un tipo de datos ya existente que restringe los valores que pueden tener las variables de dicho tipo. Se declara empleando la palabra reservada *subtype*. Así, un día laborable es en realidad un día de la semana pero ha de ser un día comprendido en el intervalo de lunes a viernes. Esto lo podemos expresar en Ada como sigue:

```
type TipoDiaSem is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
subtype TipoDiaLab is TipoDiaSem range Lun..Vie;
```

A partir de esta declaración para *TipoDiaLab* tenemos el nuevo nombre de tipo *TipoDiaLab* y podemos escribir

```
diareunion: TipoDiaLab;
```

para declarar, por ejemplo, una variable para el día de una reunión. Dicha variable sólo podrá tomar como valor días comprendidos entre *Lun* y *Vie* (inclusive ambos). Eso es lo que quiere decir el rango *Lun..Vie* en la declaración del subtipo: la lista de días comprendidos entre *Lun* y *Vie* incluyendo ambos extremos del intervalo. Por cierto, al tipo empleado para definir el subtipo se le llama **tipo base** del subtipo. Por ejemplo, decimos que *TipoDiaSem* es el tipo base de *TipoDiaLab*. Dicho de otro modo: los días laborables son días de la semana.

Dado que una variable de tipo *TipoDiaLab* es también del tipo *TipoDiaSem* ambos tipos se consideran compatibles y podemos asignar valores y variables de ambos tipos entre sí. Por ejemplo:

```
diareunion := Lun;
```

Eso sí, si en algún momento intentamos ejecutar algo como

```
diareunion := Dom;
```

el programa sufrirá un error en tiempo de ejecución en cuanto se intente realizar la asignación. Su ejecución se detendrá tras imprimir un mensaje de error informando del problema. ¿Qué otra cosa te gustaría que pasara si te ponen una reunión un domingo?

Igualmente podemos definir **subrangos** o subtipos de los enteros. Por ejemplo, para el día del mes podríamos definir

```
subtype TipoDiaDelMes is Integer range 1..31;
```

De ese modo, si utilizamos variables de tipo *TipoDiaDelMes* en lugar de variables de tipo *Integer*, el lenguaje nos protegerá comprobando que cada vez que asignemos un valor a alguna de estas variables el valor está dentro del rango permitido.

La posibilidad de definir subrangos está presente para todos los tipos enumerados. Luego podemos definir subtipos como siguen:

```
subtype TipoLetraMay is Character range 'A'..'Z';
subtype TipoLetraMin is Character range 'a'..'z';
subtype TipoDigito is Character range '0'..'9';
```

En realidad ya hemos conocido los subrangos. Cuando hablamos de la sentencia *case* vimos que una de las posibilidades a la hora de declarar un conjunto de valores era utilizar un rango, como en

```
case c is
when 'A'..'Z' =>
       ...
when '0'..'9' =>
       ...
end case;
```

La idea es la misma. También podemos utilizar el operador *in* para ver si un valor de un tipo enumerado está en un rango, como en

```
if valor in 0..9 then
       ...
end if;
```

Por último, conviene mencionar que Ada ya tiene predefinidos varios subtipos de los enteros como sigue:

```
subtype Natural is Integer range 0..Integer'Last;
subtype Positive is Integer range 1..Integer'Last;
```

Lo cual quiere decir que a partir de este momento podemos (osea, *debemos*) utilizar estos subtipos para números que sabemos que van a ser números naturales o números positivos.

6.3. Registros y tuplas

Los tipos utilizados hasta el momento tienen elementos simples, definidos por un único valor. Se dice que son **tipos elementales** o tipos simples. ¿Pero qué hacemos si nuestro programa ha de manipular objetos que no son simples? Necesitamos también los llamados **tipos compuestos**, construidos a partir de los tipos simples.

Por ejemplo, un programa que manipula cartas debe tener probablemente constantes y variables que se refieren a cartas. ¿Qué tipo ha de tener una carta? Anteriormente realizamos un programa que manipulaba ecuaciones de segundo grado y soluciones para ecuaciones de segundo grado. Terminamos declarando cabeceras de procedimiento como

```
procedure CalcularSolucion(a: Float; b: Float; c: Float;
                    sol1: out Float; sol2: out Float)
```

cuando en realidad habría sido mucho más apropiado poder declarar:

```
procedure CalcularSolucion(ec: TipoEcuacion; sol: out TipoSolucion)
```

Justo en ese programa, las declaraciones de variables del programa principal eran estas:

```
a: Float;
b: Float;
c: Float;
sol1: Float;
sol2: Float;
```

Pero en realidad habríamos querido declarar esto:

```
ec: TipoEcuacion;
sol: TipoSolucion;
```

¿Qué es una carta? ¿Qué es una ecuación de segundo grado? ¿Y una solución para dicha ecuación? Una carta es un palo y un valor. Una ecuación son los coeficientes *a*, *b* y *c*. Una solución son dos valores, digamos x_1 y x_2. Todas estas cosas son **tuplas**, o elementos pertenecientes a un producto cartesiano.

El concepto de producto cartesiano define un nuevo conjunto de elementos a partir de conjuntos ya conocidos. Por ejemplo, una carta es en realidad un par (*valor, palo*) y forma parte del producto cartesiano *valores* x *palos*.

En Ada y otros muchos lenguajes es posible definir estos productos cartesianos declarando tipos de datos conocidos como **registros** o **records**.

Vamos a ver algunos ejemplos. Aunque sería más apropiado utilizar un nuevo tipo enumerado para el caso de la baraja española, supondremos que nuestra baraja tiene las cartas numeradas de uno a diez. Luego podemos definir un nuevo tipo como sigue:

```
subtype TipoValorCarta is Integer range 1..10;
```

Igualmente podríamos definir un tipo enumerado para el palo de la baraja al que pertenece una carta:

```
type TipoPalo is (Oros, Bastos, Copas, Espadas);
```

Ahora podemos definir un tipo nuevo para una pareja de ordenada compuesta por un valor y un palo. La idea es la misma que cuando construimos tuplas con un producto cartesiano en matemáticas.

```
type TipoCarta is
record
       valor: TipoValorCarta;
       palo: TipoPalo;
end record;
```

Como puede verse, la declaración utiliza la palabra reservada *record* (de ahí el nombre de estos tipos de datos) e incluye entre *record* y *end record* una declaración para cada elemento de la tupla. En este caso, una carta está formada por algo llamado *valor* (de tipo *TipoValorCarta*) y algo llamado *palo* (de tipo *TipoPalo*). ¡Y en este orden!

Luego algo de tipo *TipoCarta* es en realidad una tupla formada por algo de tipo *TipoValorCarta* y algo de tipo *TipoPalo.* Eso es razonable. El ‘‘2 de bastos’’ es en realidad algo definido por el ‘‘2’’ y por los ‘‘bastos’’.

Para una ecuación de segundo grado podríamos definir:

```
-- a * x ** 2 + b * x + c = 0
type TipoEcuacion is
record
       a: Float;
       b: Float;
       c: Float;
end record;
```

En este caso una ecuación es un triplete ordenado de tres cosas (todas de tipo *Float*). La primera es algo llamado *a*, la segunda *b* y la tercera *c*.

Cada elemento del *record* se denomina **campo** del *record*. Así, una carta será un *record* con dos campos: un campo denominado *valor* y otro campo denominado *palo*. Es importante saber que estos campos mantienen siempre el mismo orden. En una carta tendremos primero el valor y después el palo. En la memoria del ordenador una variable (o un valor) de tipo *TipoCarta* tiene el aspecto de dos variables guardadas una tras otra: una primero de tipo *TipoValorCarta* y otra después de tipo *TipoPalo*. La figura 6.2 muestra el aspecto de dos *records* en la memoria del ordenador.

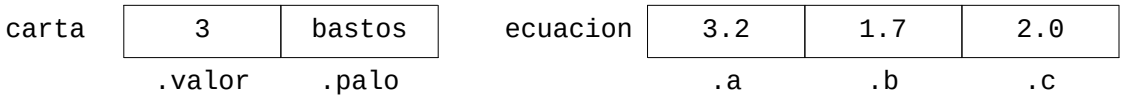


Figura 6.2: *Aspecto de los records de los ejemplos en la memoria del ordenador.*

¿Y cómo podemos utilizar una variable de tipo *TipoCarta*? En general, igual que cualquier otra variable. Por ejemplo:

```
carta1: TipoCarta;
carta2: TipoCarta;
       ...
carta1 := carta2;
```

No obstante, hay diferencias entre los registros y el resto de los tipos que hemos visto hasta el momento en cuanto a qué operaciones se pueden hacer con ellos:

- Es posible utilizar *records* como parámetros de funciones y procedimientos y como objetos devueltos por funciones. En particular, es posible asignarlos entre sí (se asignan los campos uno por uno, como cabría esperar).
- Es posible comparar *records* (del mismo tipo) con ‘‘`=`’’ y con ‘‘`/=`’’. Dos *records* se consideran iguales si sus campos son iguales dos a dos y se consideran distintos en caso contrario.
- **No** es posible emplear ningún otro operador de comparación a los *records*.
- **No** es posible ni leer ni escribir *records*. Para leer registros tenemos que leer campo por campo y para escribirlos tenemos que escribir campo por campo.

Necesitamos poder referirnos a los campos de un *record* de forma individual. Bien para consultar su valor, bien para actualizarlo. Eso se puede hacer con la llamada **notación punto**. Para referirse al campo de un *record* basta utilizar el nombre del *record* (de la variable o constante de ese tipo) y escribir a continuación un ‘‘`.`’’ y el nombre del campo que nos interesa. Por ejemplo, considerando *carta1*, de *TipoCarta*, podríamos decir:

```
carta1.valor := 1;
carta1.palo := Bastos;
```

Esto haría que *carta1* fuese el as de bastos. La primera sentencia asigna el valor *1* al campo *valor* de *carta1*.

Podemos utilizar *carta1.valor* en cualquier sitio donde podamos utilizar una variable de tipo *TipoValorCarta*. Igual sucede con cualquier otro campo. Un campo se comporta como una variable del tipo al que pertenece (el campo).

Recuerda que un *record* es simplemente un tipo de datos que agrupa en una sola cosa varios elementos, cada uno de su propio tipo de datos. En la memoria del ordenador una variable de tipo *record* es muy similar a tener juntas varias variables (una por cada campo, como muestra la figura 6.2). No obstante, a la hora de programar la diferencia es abismal. Es infinitamente más fácil manipular cartas que manipular parejas de variables que no tienen nada que ver entre sí. Para objetos más complicados la diferencia es aún mayor.

Al declarar constantes de tipo *record* es útil la sintaxis que tiene el lenguaje para expresar **agregados**. Un agregado es simplemente una secuencia ordenada de elementos que se consideran agregados para formar un elemento más complejo. Esto se verá claro con un ejemplo; este es el as de copas:

```
AsCopas: constant TipoCarta := (1, Copas);
```

Como puede verse, hemos inicializado la constante escribiendo entre paréntesis una expresión para cada campo del registro, en el orden en que están declarados los campos. El ‘‘`1`’’ se asigna a *AsCopas.valor* y ‘‘*Copas*’’ se asigna a *AsCopas.palo* (como valores iniciales, durante la declaración de la constante). La tupla escrita a la derecha de la asignación es lo que se denomina agregado.

Veamos otro ejemplo para seguir familiarizándonos con los *records*. Este procedimiento lee una carta. Como tiene efectos laterales al leer de la entrada no podemos implementarlo con una función.

```
1  procedure GetCarta(carta: out TipoCarta) is
2  begin
3      Get(carta.valor);
4      Get(carta.palo);
5  end;
```

Dado que ambos campos son de tipos elementales (no son de tipos compuestos de varios elementos, como los records) podemos leerlos directamente.

6.4. Abstracción

Es muy útil podernos olvidar de lo que tiene dentro un *record*; del mismo modo que resulta muy útil podernos olvidar de lo que tiene dentro un subprograma.

Si hacemos un programa para jugar a las cartas, gran parte del programa estará manipulando variables de tipo carta sin prestar atención a lo que tienen dentro. Por ejemplo, repartiéndolas, jugando con ellas, pasándolas de la mano del jugador a la mesa, etc. Igualmente, el programa estará continuamente utilizando procedimientos para repartirlas, para jugarlas, para pasarlas a la mesa, etc. Si conseguimos hacer el programa de este modo, evitaremos tener que considerar todos los detalles del programa al programarlo. *Esta es la clave para hacer programas: abstraer y olvidar los detalles.*

Los *records* nos permiten abstraer los datos que manipulamos (verlos como elementos abstractos en lugar de ver sus detalles) lo mismo que los subprogramas nos permiten abstraer los algoritmos que empleamos (verlos como operaciones con un nombre en lugar de tener que pensar en los detalles del algoritmo). Si tenemos que tener en la mente todos los detalles de todos los algoritmos y datos que empleamos, ¡Nunca haremos ningún programa decente que funcione! Al

menos ninguno que sea mucho más útil que nuestro ‘‘hola π’’. Si abstraemos... ¡Seremos capaces de hacer programas tan elaborados como sea preciso!

Igual que las construcciones como el *if-then* (y otras que veremos) permiten estructurar el código del programa (el flujo de control), con las construcciones como *record* (y otras que veremos) podemos estructurar los datos. Por eso normalmente se habla de **estructuras de datos** en lugar de hablar simplemente de datos.

Los datos están estructurados en tipos de datos complejos hechos a partir de tipos de datos más simples; hechos a su vez de tipos mas simples... hasta llegar a los tipos básicos del lenguaje. Podemos tener *records* cuyos campos sean otros *records*, etc. Estructurar los datos utilizando tipos de datos compuestos de tipos más simples es similar a estructurar el código de un programa utilizando procedimientos y funciones.

Recuerda lo que dijimos hace tiempo: *Algoritmos + Estructuras = Programas.*

6.5. Geometría

Queremos manipular figuras geométricas en dos dimensiones. Para manipular puntos podríamos declarar:

```
type TipoPunto is
record
       x: Integer;
       y: Integer;
end record;
```

Un punto es un par de coordenadas, por ejemplo, en la pantalla del ordenador. Podríamos declarar el origen de coordenadas como

```
Origen: constant TipoPunto := (0, 0);
```

lo que haría que *Origen.x* fuese cero y que *Origen.y* fuese cero.

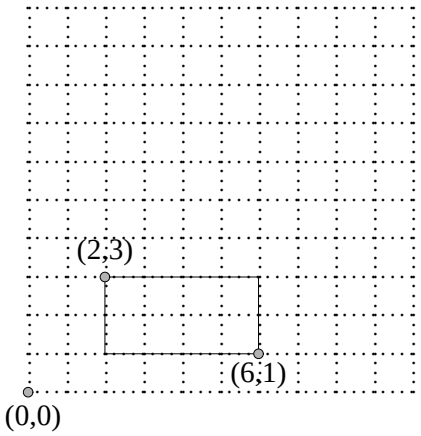


Figura 6.3: *Coordenadas en dos dimensiones incluyendo origen y un rectángulo.*

Si ahora queremos representar un rectángulo, podríamos hacerlo definiendo el punto de arriba a la izquierda y el punto de abajo a la derecha en el rectángulo, como muestra la figura 6.3. Podríamos llamar a ambos puntos el menor y el mayor punto del rectángulo y declarar un nuevo tipo como sigue:

```
type TipoRectangulo is
record
       menor: TipoPunto;
       mayor: TipoPunto;
end record;
```

Ahora, dado un rectángulo

```
r: TipoRectangulo;
```

podríamos escribir *r.menor* para referirnos a su punto de arriba a la izquierda. O quizá *r.menor.y* para referirnos a la coordenada *y* de su punto de arriba a la izquierda.

Un círculo podríamos definirlo a partir de su centro y su radio. Luego podríamos declarar algo como

```
type TipoCirculo is
record
       centro: TipoPunto;
       radio: Integer;
end record;
```

Las elipses son similares, pero tienen un radio menor y un radio mayor:

```
type TipoElipse is
record
       centro: TipoPunto
       radiomin: Integer;
       radiomax: Integer;
end record;
```

Podemos centrar una elipse en un círculo haciendo algo como

```
elipse.centro := circulo.centro;
```

lo que modifica el centro de la elipse (que es de *TipoPunto*) para que tenga el mismo valor que el del centro del círculo. O podemos quizá ver si una elipse y un círculo están centrados en el mismo punto:

```
if elipse.centro = circulo.centro then
       ...
end if;
```

Al comparar los campos *centro* de ambos *records* estamos en realidad comparando dos *records* de tipo *TipoPunto*. Dichos *records* serán iguales si sucede que el campo *x* es igual el ambos *records* y el campo *y* es igual en ambos *records.*

Vamos ahora a inscribir un círculo en un cuadrado tal y como muestra la figura 6.4. Necesitamos definir tanto el centro del círculo como su radio:

```
circulo.centro.x := (cuadrado.min.x + cuadrado.max.x) / 2;
circulo.centro.y := (cuadrado.min.y + cuadrado.max.y) / 2;
circulo.radio := (cuadrado.max.x - cuadrado.min.x) / 2;
```

Hemos supuesto que nuestro *cuadrado* es en realidad un cuadrado. ¿Hemos perdido el juicio? Puede ser, pero es que nuestra variable *cuadrado* es de tipo *TipoRectangulo*. Por supuesto, dado su nombre, debería ser un cuadrado. ¿Cómo sabemos si es un cuadrado? Con esta expresión de tipo *Boolean*:

```
cuadrado.max.x - cuadrado.min.x = cuadrado.max.y - cuadrado.min.y
```

Confiamos en que la mecánica de declaración de records y el uso elemental de los mismos, para operar con ellos y para operar con sus campos, se entienda mejor tras ver estos ejemplos. Ahora

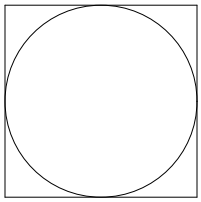


Figura 6.4: *Círculo inscrito en un cuadrado.*

vamos a hacer programas con todo lo que hemos visto.

6.6. Aritmética compleja

Queremos sumar, restar multiplicar y dividir números complejos. Para empezar, podríamos definir un tipo de datos para un número complejo (en coordenadas cartesianas):

```
-- un numero complejo en coordenadas cartesianas
type  TipoComplejo is
record
       re: Float;    -- parte real
       im: Float;    -- parte imaginaria
end record;
```

Antes siquiera de pensar en cuál es el subproblema más pequeño por el que podemos empezar, considerando que tenemos un nuevo tipo de datos, parece que podríamos definir una función para construir un nuevo valor de tipo *TipoComplejo*, un procedimiento para leer un número complejo de la entrada y otro para escribir un número complejo en la salida. Nuestro programa quedaría como sigue. Nótese el tipo de paso de parámetros que hemos empleado en cada ocasión. Sólo *LeerComplejo* emplea paso de parámetros por referencia (concretamente, de salida).

complejos.adb

```
1     --
2     -- Aritmetica de numeros complejos
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure Complejos is

10          -- un numero complejo en coordenadas cartesianas
11          type  TipoComplejo is
12          record
13               re: Float; -- parte real
14               im: Float; -- parte imaginaria
15          end record;

17          -- necesario para leer/escribir reales
18          package FloatIO is new Float_IO(Float);
19          use FloatIO;
```

```
21          function NuevoComplejo(re: Float; im: Float) return TipoComplejo is
22               c: TipoComplejo;
23          begin
24               c.re := re;
25               c.im := im;
26               return c;
27          end;

29          procedure LeerComplejo(c: out TipoComplejo) is
30               re: Float;
31               im: Float;
32          begin
33               Put("Introduce parte real e imaginaria: ");
34               Get(re);
35               Get(im);
36               c := NuevoComplejo(re, im);
37          end;

39          -- escribirlo como "re + im i"
40          procedure EscribirComplejo(c: TipoComplejo) is
41          begin
42               Put(c.re);
43               Put(" + ");
44               Put(c.im);
45               Put( " i");
46          end;

48          c: TipoComplejo;
49     begin
50          LeerComplejo(c);
51          EscribirComplejo(c);
52     end;
—
```

Antes de seguir, como de costumbre, debemos ejecutar el programa para ver si todo funciona. Y parece que sí.

```
; ada complejos.adb
; complejos
Introduce parte real e imaginaria: 3.4 2
 3.40000E+00 +  2.00000E+00 i
```

Hemos de recordar que en Ada podríamos escribir algo como

```
c := (re, im);
```

para inicializar un número complejo *c* a partir de su parte real y su parte imaginaria. Pero no queremos saber nada de los detalles respecto a cómo está hecho un número complejo. Por eso hemos definido una función llamada *NuevoComplejo* para crear un número complejo. Nos gustaría que nuestro programa manipulase los números complejos utilizando las operaciones que definamos, del mismo modo que hacemos con los enteros.

Sumar y restar números complejos es fácil: sumamos y restamos las partes reales e imaginarias. Ambas operaciones podrían ser funciones que devuelvan nuevos valores complejos con el resultado de la operación.

```
function Sumar(c1: TipoComplejo; c2: TipoComplejo) return TipoComplejo is
begin
       return NuevoComplejo(c1.re + c2.re, c1.im + c2.im);
end;

function Restar(c1: TipoComplejo; c2: TipoComplejo) return TipoComplejo is
begin
       return NuevoComplejo(c1.re - c2.re, c1.im - c2.im);
end;
```

Podríamos continuar así definiendo funciones para la multiplicación, división, conjugado, etc. Una vez programadas estas operaciones cualquier programa que manipule números complejos puede olvidarse por completo de cómo están hechos éstos. Y si descubrimos que en un programa determinado necesitamos una operación que se nos había olvidado, la definimos. Por ejemplo, es muy posible que necesitemos funciones para obtener la parte real y la parte imaginaria de un número complejo. Una alternativa sería utilizar *c.re* y *c.im*, lo que es más sencillo (aunque rompe la abstracción puesto que vemos cómo está hecho un número complejo).

6.7. Cartas del juego de las 7½

Queremos hacer un programa para jugar a las 7½ y, por el momento, queremos un programa que lea una carta de la entrada estándar y escriba su valor según este juego. Este es un juego para la baraja española donde las figuras valen ½ y el resto de cartas su valor nominal.

Vamos a hacer con los datos lo mismo que hemos hecho con los subprogramas: ¡Suponer que ya los tenemos! En cuanto sean necesarios, eso sí. Por ejemplo, parece que claro que lo debemos hacer es:

1 Leer una carta

2 Calcular su valor

3 Imprimir su valor.

Pues entonces... ¡Hacemos justo eso!

```
procedure Carta is

       ...

       carta: TipoCarta;
       valor: Float;
begin
       LeerCarta(carta);
       valor := ValorDeCarta(carta);
       Put(valor);
end;
```

Dado que un valor es un número real en este caso (hay cartas que valen ½) sabemos que *valor* debe ser un *Float* y que podemos utilizar *Put* para escribir el valor.

Hemos hecho casi lo mismo con la carta. ¡Aunque Ada no tiene ni idea de lo que es una carta! Nos hemos inventado el tipo *TipoCarta* y hemos declarado una variable de ese tipo. Como queremos leer una carta, nos hemos inventado *LeerCarta* (que necesita que le demos la variable cuyo valor hay que leer). Igualmente, *ValorDeCarta* es una función que también nos hemos sacado de la manga para que nos devuelva el valor dada una carta. ¡Fácil! ¿no?

Antes de escribir los procedimientos, resulta muy útil definir el tipo de datos (muchas veces el tipo que empleemos determinará que aspecto tiene la implementación de los procedimientos y funciones que lo manipulen). Una carta es un palo y un valor. Pues nos inventamos dos tipos, *TipoPalo* y *TipoValor*, y declaramos:

```
type TipoCarta is
record
       valor: TipoValor;
       palo: TipoPalo;
end record;
```

Y ahora tenemos que declarar en el programa, más arriba, estos tipos que nos hemos inventado:

```
type TipoValor is (As, Dos, Tres, Cuatro, Cinco,
                   Seis, Siete, Sota, Caballo, Rey);

type TipoPalo is (Oros, Copas, Bastos, Espadas);
```

Así va surgiendo el programa. Un procedimiento para leer una carta necesitará leer el palo y leer el valor; y tendrá un parámetro pasado por referencia (de salida en realidad) para la carta.

```
1  procedure LeerCarta(carta: out TipoCarta) is
2      valor: TipoValor;
3      palo: TipoPalo;
4  begin
5      Put("valor :");
6      Get(valor);
7      Put("palo: ");
8      Get(palo);
9      carta := NuevaCarta(valor, palo);
10     New_Line;
11 end;
```

Como resulta que tanto *valor* como *palo* son de tipos enumerados, podemos particularizar *Enumeration_IO* para utilizar *Get* para leerlos. Luego ahora añadimos al programa...

```
package Valor_InOut is new Enumeration_IO(TipoValor);
use Valor_InOut;
package Palo_InOut is new Enumeration_IO(TipoPalo);
use Palo_InOut;
```

Como podrás ver, nos hemos inventado una función *NuevaCarta* que construya y devuelva una nueva carta dados su valor y su palo. Podríamos haber escrito

```
carta := (valor, palo);
```

pero resultaba mucho más sencillo aplazar cómo construir una carta. Y de paso el código se lee algo mejor. Esta función es simplemente:

```
1  -- Construir una carta
2  function NuevaCarta(valor: TipoValor; palo: TipoPalo) return TipoCarta is
3      carta: TipoCarta;
4  begin
5      carta.valor := valor;
6      carta.palo := palo;
7      return carta;
8  end;
```

Para tipos de datos más complicados esta costumbre de utilizar funciones (o procedimientos) para crear o inicializar nuevos elementos del tipo resultará muy útil.

¿Cómo implementamos el subprograma que nos da el valor de una carta? Claramente es una función: le damos una carta y nos devuelve su valor. Bueno, todo depende de si la carta es una figura o no. Si lo es, entonces su valor es 0.5; en otro caso su valor es el valor del número de la carta. Podemos entonces escribir esto por el momento...

```
function ValorCarta(carta: TipoCarta) return Float is
    valor: Integer;
begin
    if EsFigura(carta) then
          return 0.5;
    else
          return 0;-- XXX esto falta XXX
    end if;
end;
```

De nuevo, nos hemos inventado *EsFigura*. Se supone que esa función devolverá *True* cuando tengamos una carta que sea una figura. Fíjate en que todo el tiempo tratamos de conseguir que el programa manipule cartas u otros objetos que tengan sentido en el problema en que estamos trabajando. Si se hace esto así, la estructura de los datos y la estructura del programa saldrá sola. Es curioso que los escultores afirmen que lo mismo sucede con las esculturas, que salen solas de las piedras en que se esculpen.

¿Cuál es el valor numérico de una carta? Bueno, necesitamos que el *As* valga 1, el *Dos* valga 2, etc. Si tomamos la posición del valor de la carta y le restamos la posición del *As* en el tipo enumerado entonces conseguimos un 0 para el *As*, un 1 para el *Dos*, etc. Por lo tanto necesitamos sumar uno a esta resta y podemos escribir:

```
-- Valor de una carta en las 7 y 1/2
function ValorCarta(carta: TipoCarta) return Float is
    valor: Integer;
    pos: Integer;
begin
    if EsFigura(carta) then
          return 0.5;
    else
          pos := TipoValor'Pos(carta.valor);
          valor :=  pos - TipoValor'Pos(As) + 1;
          return Float(valor);
    end if;
end;
```

¡No hemos dudado en declarar *valor* y *pos*! Los hemos declarado con el único propósito de escribir la expresión

```
Float(TipoValor'Pos(carta.valor) - TipoValor'Pos(As) + 1)
```

poco a poco. Primero tomamos el valor de la carta... su posición en el enumerado... calculamos el valor a partir de ahí... y lo convertimos a un número real.

Nos falta ver si una carta es una figura. Para esto podríamos simplemente escribir:

```
function EsFigura(carta: TipoCarta) return Boolean is
begin
    return carta.valor >= Sota and carta.valor <= Rey;
end;
```

Recuerda que lo que es (o no es) una figura es la carta, no el valor de la carta. Por eso nuestra función trabaja con una carta y no con un valor.

Y ya está. El programa está terminado.

carta.adb

```
   --
   -- Ver el valor de una carta en las 7 y 1/2.
   --
```

```
5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure Carta is

10          -- Carta de la baraja española
11          type TipoValor is (As, Dos, Tres, Cuatro, Cinco,
12                             Seis, Siete, Sota, Caballo, Rey);

14          type TipoPalo is (Oros, Copas, Bastos, Espadas);

16          type TipoCarta is
17          record
18               valor: TipoValor;
19               palo: TipoPalo;
20          end record;

22          package Valor_InOut is new Enumeration_IO(TipoValor);
23          use Valor_InOut;
24          package Palo_InOut is new Enumeration_IO(TipoPalo);
25          use Palo_InOut;
26          package Float_InOut is new Float_IO(Float);
27          use Float_InOut;

29          function NuevaCarta(valor: TipoValor; palo: TipoPalo) return TipoCarta is
30               carta: TipoCarta;
31          begin
32               carta.valor := valor;
33               carta.palo := palo;
34               return carta;
35          end;

37          function EsFigura(carta: TipoCarta) return Boolean is
38          begin
39               return carta.valor in Sota..Rey;
40          end;

42          -- Valor de una carta en las 7 y 1/2
43          function ValorCarta(carta: TipoCarta) return Float is
44               valor: Integer;
45               pos: Integer;
46          begin
47               if EsFigura(carta) then
48                    return 0.5;
49               else
50                    pos := TipoValor'Pos(carta.valor);
51                    valor :=  pos - TipoValor'Pos(As) + 1;
52                    return Float(valor);
53               end if;
54          end;
```

```
56          procedure LeerCarta(carta: out TipoCarta) is
57               valor: TipoValor;
58               palo: TipoPalo;
59          begin
60               Put("valor :");
61               Get(valor);
62               Put("palo: ");
63               Get(palo);
64               carta := NuevaCarta(valor, palo);
65               New_Line;
66          end;

68          carta: TipoCarta;
69          valor: Float;
70     begin
71          LeerCarta(carta);
72          valor := ValorCarta(carta);
73          Put("La carta vale ");
74          Put(valor);
75          New_Line;
76     end;
—
```

Aunque no lo hemos mencionado, hemos compilado y ejecutado el programa en cada paso de los que hemos descrito. Si es preciso durante un tiempo tener

```
type TipoCarta is Integer;
```

para poder declarar cartas, pues así lo hacemos. O, si necesitamos escribir un

```
return 0;
```

en una función para no escribir su cuerpo, pues también lo hacemos. Lo que importa es poder ir probando el programa poco a poco.

Si el problema hubiese sido mucho más complicado. Por ejemplo, ‘‘jugar a las 7½’’, entonces habríamos simplificado el problema todo lo posible para no abordarlo todo al mismo tiempo. Muy posiblemente, dado que sabemos que tenemos que manipular cartas, habríamos empezado por implementar el programa que hemos mostrado. Podríamos después complicarlo para que sepa manejar una mano de cartas y continuar de este modo hasta tenerlo implementado por completo. La clave es que en cada paso del proceso sólo queremos tener un trozo de código (o datos) nuevo en el que pensar.

6.8. Variaciones

En ocasiones (aunque no con frecuencia) necesitaremos un único tipo de datos para objetos que, aunque del mismo tipo, tienen diferencias entre sí.

Por ejemplo, un programa que manipule figuras geométricas podría querer definir un único tipo de datos para una figura geométrica y algunas operaciones básicas sobre él. Algo parecido a...

```
type TipoFigura is ... ;

procedure DibujarEnPosicion(pos: TipoPunto; fig: TipoFigura);

function AreaOcupada(fig: TipoFigura) return TipoRectangulo;

procedure RellenarFigura(fig: TipoFigura);

 ...
```

En realidad las operaciones son lo de menos en este momento. Pero parece interesante poder tener un único conjunto de operaciones para todas las figuras y un único tipo de datos para representar a todas las figuras (dado que tienen mucho en común).

Queremos declarar un *record* para nuestras figuras. Parece razonable que dicho *record* tenga por lo menos un campo para identificar la posición en la que se encuentra la figura. Tal vez queramos incluir también un campo para indicar si hay que dibujar la figura con o sin relleno y otro campo para indicar el color del relleno. Resultaría algo como esto:

```
type TipoFigura is
record
       pos: TipoPunto;
       estarellena: Boolean;
       relleno: TipoColor;
       ...
end record;
```

No obstante, un círculo sólo necesita otro campo: un radio. Un rectángulo necesitaría un valor para la longitud horizontal y otro para la vertical (suponiendo que lo centramos en *pos*). Y así cada figura distinta necesitaría un conjunto de campos distinto.

Otro ejemplo de esta situación podemos tomarlo de una aplicación para gestionar información médica de personas. Necesitaremos un tipo llamado, tal vez, *TipoPaciente*. Este tipo de datos necesitará al menos un número de la seguridad social. Algo similar a

```
type TipoPaciente is
record
       nss: Integer;
       ...
end record;
```

Ahora bien, para hombres podría interesar un campo que indique si tienen barba (hay que afeitarlos antes de operarlos). Para mujeres en cambio podría interesar otro que indique si han tenido partos.

Lo que estamos necesitando es un *record* cuyos campos varíen en función del valor de un campo utilizado como **discriminante** (el tipo de figura en el primer ejemplo y el sexo en el segundo ejemplo). A este tipo de *records* se los conoce como **records con variantes**.

Para el primer ejemplo podríamos declarar en Ada:

```
type TipoClaseFigura is (Ninguna, Circulo, Cuadrado, Rectangulo);

type TipoFigura(clase: TipoClaseFigura := Ninguna) is
record
       pos: TipoPunto;
       estarellena: Boolean;
       relleno: TipoColor;
       case clase is
       when Ninguna =>
              null;
       when Circulo =>
              radio: Integer;
       when Cuadrado =>
              lado: Integer;
       when Rectangulo =>
              ancho: Integer;
              alto: Integer;
       end case;
end record;
```

Para el segundo ejemplo podríamos declarar

```
type TipoClasePaciente is (Ninguno, Hombre, Mujer);

type TipoPaciente(sexo: TipoSexo := Ninguno) is
record
       nss: Integer;
       case sexo is
       when Ninguno =>
              null;
       when Hombre =>
              conbarba: Boolean;
       when Mujer =>
              numeropartos: Integer;
       end case;
end record;
```

La idea que es un objeto de *TipoFigura* es algo que *siempre* tiene los campos: *clase*, *pos*, *estarellena* y *relleno*. Ahora bien, el resto de campos dependen del valor que tengamos en *clase*. Por eso hemos definido también un tipo enumerado para identificar las clases de figuras que tenemos.

Un círculo tendrá en la memoria del ordenador el aspecto de un record declarado como

```
record
       clase: TipoClaseFigura;
       pos: TipoPunto;
       estarellena: Boolean;
       relleno: TipoColor;
       radio: Integer;
end record;
```

Y en cambio, un rectángulo tendrá en la memoria el aspecto de un *record* declarado como

```
record
       clase: TipoClaseFigura;
       pos: TipoPunto;
       estarellena: Boolean;
       relleno: TipoColor;
       ancho: Integer;
       alto: Integer;
end record;
```

Como puede verse el tamaño de una figura en la memoria será el del mayor de estos *records* (La figura 6.5 ilustra esto para el caso de una variable *paciente* de *TipoPaciente*). El tamaño utilizado por un registro con variantes es el tamaño de la variante más grande.

Como puede suponerse, una vez inicializada una variable para que sea de una clase concreta de figura no debe cambiarse el tipo ni puede accederse a ningún campo definido para otras clases de figura. Por ejemplo, si inicializamos una figura para que sea un círculo entonces esta será un círculo durante toda su vida; y sólo tendrá los campos de un círculo.

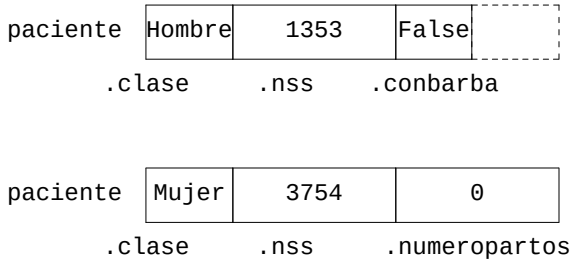


Figura 6.5: *Aspecto en la memoria de dos variantes del mismo tipo de record.*

Veamos ahora cómo se utilizan los registros con variantes. Esto es mejor hacerlo con ejemplos. Esta sería una declaración para un paciente sin definir, por ejemplo:

```
paciente: TipoPaciente;
```

¡Fácil! ¿no? Tras la declaración el paciente es, por el momento, de *sexo Ninguno*. Fíjate en que en la declaración de *TipoPaciente* hemos escrito entre paréntesis la declaración del campo que discrimina entre una variante y otra, y hemos definido un valor inicial para dicho campo:

```
type TipoPaciente(sexo: TipoSexo := Ninguno) is ...
```

Como para esa variante utilizamos la palabra reservada *null* para indicar que no existe ningún campo adicional, los únicos campos disponibles por el momento son *sexo* y *nss*.

Ahora al vez querríamos hacer que *paciente* corresponda a un hombre. Podríamos, por ejemplo, inicializarlo como sigue:

```
paciente := (Hombre, 12354, False);
```

Tenemos que inicializar *todos* los campos en la misma asignación. Esto se hace así para que no exista posibilidad de error en cuanto a qué variante estamos creando y en cuanto a qué campos tiene. Desde este momento *paciente.sexo* es *Hombre*; esto implica que tenemos el campo *conbarba* además de los dos campos comunes a todas las variantes.

Si intentamos asignar algo al campo *numeropartos* y tenemos suerte, se producirá un error. Si no la tenemos, estaremos modificando el valor de *conbarba*, dado que en realidad tenemos un *record* sólo con los campos para la variante que queremos, pero no para todas a la vez.

El aspecto de las operaciones que manipulan *records* con variantes suele ser una sentencia *case* que, en función de la variante concreta, implementa la operación de un modo u otro. Por ejemplo:

```
procedure RellenarFigura(fig: TipoFigura) is
begin
       case fig.clase is
       when Ninguna =>
              null;
       when Circulo =>
              ... rellenar el circulo ...
       when Cuadrado =>
              ... rellenar el cuadrado ...
       when Rectangulo =>
              ... rellenar el rectangulo ...
       end case;
end;
```

En cada rama del *case* podemos utilizar sin problemas los campos de la variante concreta a que nos referimos. Fuera del *case* sólo podemos utilizar los campos comunes a todas las variantes (puesto que no sabemos qué clase de variante tenemos entre manos).

El uso de este tipo de records suele ser complicado y se sugiere no practicar con el hasta dominar el uso de records convencionales.

Problemas

Obviamente, en todos los problemas se exige utilizar tipos enumerados y registros allí donde sea apropiado (en lugar de hacerlos como hemos hecho en capítulos anteriores). Como de costumbre hay enunciados para problemas cuya solución ya tienes; deberías hacerlos ahora sin mirar la solución.

1 Ver que día de la semana será mañana dado el de hoy.

2 Calcular los días del mes, dado el mes.

3 Calcular el día del año desde comienzo de año dado el mes y el día del mes.

4 Calcular el día de la semana, suponiendo que el día uno fue Martes, dado el número de un día.

5 Leer y escribir el valor de una carta utilizando también un tipo enumerado para el valor, para que podamos tener sota, caballo, rey y as.

6 Dada una fecha, y suponiendo que no hay años bisiestos y que todos los meses son de 30 días, calcular el número de días desde el 1 enero de 1970.

7 Ver la distancia en días entre dos fechas con las mismas suposiciones que el problema anterior.

8 Sumar, restar y multiplicar números complejos.

9 Ver si un punto está dentro de un rectángulo en el plano.

10 Mover un rectángulo según un vector expresando como un punto.

11 Ver si una carta tiene un valor mayor que otra en el juego de las 7 y media.

12 Decidir si un punto está dentro de un circulo.

13 Evaluar expresiones aritméticas de dos operandos y un sólo operador para sumas, restas, multiplicaciones y divisiones.

14 Resolver los problemas anteriores que tratan de figuras geométricas de tal forma que los datos se obtengan desde la entrada y que puedan darse los puntos en cualquier orden.

15 Decidir se si aprueba una asignatura considerando que hay dos ejercicios, un test y un examen y que el test se considera apto o no apto y el examen se puntúa de 0 a 10. Para aprobar hay que superar el test y aprobar el examen.

15 Decidir si una nota en la asignatura anterior es mayor que otra.

16 Componer colores primarios (rojo, verde y azul).

17 Ver si un color del arcoiris tiene mayor longitud de onda que otro.

18 Implementar una calculadora de expresiones aritméticas simples. Deben leerse dos operandos y un operador e imprimir el valor de la expresión resultante.

19 Para cada uno de los siguientes objetos, define un tipo de datos en Ada y haz un procedimiento para construir un nuevo elemento de ese tipo, otro para leer una variable de dicho tipo y otro para escribirla. Naturalmente, debes hacer un programa para probar que el tipo y los procedimientos funcionan. Haz, por ejemplo, que dicho programa escriba dos veces el objeto leído desde la entrada.

   a) Un ordenador, para un programa de gestión de inventario de ordenadores. Un ordenador tiene una CPU dada, una cantidad de memoria instalada dada, una cantidad de disco dado y ejecuta a un número de MHz dado. Tenemos CPUs que pueden ser Core duo, Core 2 duo, Core Quad, Core i7 o Cell. La memoria se mide en números enteros de Mbytes para este problema.

   b) Una tarjeta gráfica, que puede ser ATI o Nvidia y tener una CPU que ejecuta a un número de MHz dado (Sí, las tarjetas gráficas son en realidad ordenadores empotrados en una tarjeta) y una cantidad memoria dada expresada en Mbytes.

   c) Un ordenador con tarjeta gráfica.

   d) Un empleado. Suponiendo que tenemos a Jesús, María y José como empleados inmortales en nuestra empresa y que no vamos a despedir ni a contratar a nadie más. Cada empleado tiene un número de DNI y una edad concreta. Como nuestra empresa es de líneas aéreas nos interesa el peso de cada empleado (puesto que supone combustible).

   e) Un empleado con ordenador con tarjeta gráfica.

   f) Un ordenador con tarjeta gráfica y programador (suponiendo que hemos reconvertido nuestra empresa de líneas aéreas a una empresa de software).

   g) Un ordenador con dos tarjetas gráficas (una para 2D y otra para 3D) cada una de las cuales ha sido montada por uno de nuestros empleados.

20 Muchos de los problemas anteriores de este curso se han resuelto sin utilizar ni tipos enumerados ni registros cuando en realidad deberían haberse programado utilizando éstas facilidades. Localiza dichos problemas y arreglalos.

7 — Bucles

7.1. Jugar a las 7½

El juego de las 7½ consiste en tomar cartas hasta que el valor total de las cartas que tenemos sea igual o mayor a 7½ puntos. Como mencionamos en el capítulo anterior, cada carta vale su número salvo las figuras (sota, caballo y rey), que valen ½ punto.

Ya vimos cómo implementar en un programa lo necesario para leer una carta e imprimir su valor. La pregunta es... ¿Cómo podemos implementar el juego entero? Necesitamos leer cartas e ir acumulando el valor total de las mismas hasta que dicho valor total sea mayor o igual a 7½. ¿Cómo lo hacemos?

Además de la secuencia (representada en Ada por el bloque de sentencias entre *begin* y *end*) y la bifurcación condicional (para la que tenemos en Ada todas las variantes de la sentencia *if-then* y la sentencia *case*) tenemos otra estructura de control: los **bucles**.

Un bucle permite repetir una sentencia (o varias) el número de veces que sea preciso. Recordemos que las repeticiones son uno de los componentes básicos de los programas estructurados y, en general, son necesarios para implementar algoritmos.

Existen varias construcciones para implementar bucles. El primer tipo de las que vamos a ver se conoce como **bucle while** en honor a la palabra reservada *while* que se utiliza en esta estructura. Un *while* tiene el siguiente esquema:

```
inicializacion;
while condicion loop   -- mientras ... repetir ...
      sentencias;
end loop;              -- fin repetir
```

Donde *inicialización* es una o varias sentencias utilizadas para preparar el comienzo del bucle propiamente dicho, que es el código entre *while* y *end loop*. (En muchas ocasiones no es preciso preparar nada y esta parte se omite). La *condición* determina si se ejecuta el cuerpo del bucle o no (las sentencias entre *loop* y *end loop*). El bucle se repite mientras se cumpla la condición, evaluando dicha condición antes de ejecutar el cuerpo del bucle en cada ocasión. El flujo de control sigue el esquema mostrado en la figura 7.1.

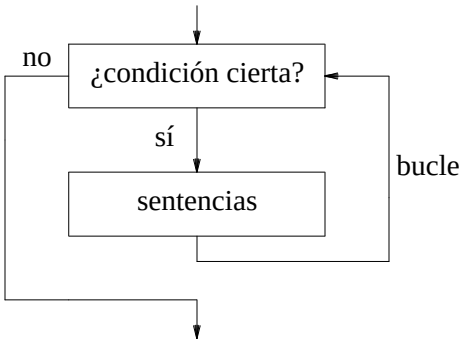


Figura 7.1: *Flujo de control en un bucle while.*

Como de costumbre, el flujo de control entra por la parte superior del bucle y continúa (quizá dando algunas vueltas en el bucle) tras el mismo: un único punto de entrada y un único punto de salida. A ejecutar un bucle se lo denomina **iterar**. A cada ejecución del cuerpo del bucle se la suele denominar **iteración**. Aunque a veces se habla de *pasada* del bucle.

Veamos cómo jugar a las 7½. Inicialmente no tenemos puntos. Y mientras el número de puntos sea menor que 7½ tenemos que, repetidamente, leer una carta y sumar su valor al total de puntos:

```
puntos := 0.0;
while puntos < 7.5 loop
       LeerCarta(carta);
       puntos := puntos + ValorCarta(carta);
end loop;
```

Cuando el programa llega a la línea del *while* se evalúa la condición. En este caso la condición es:

```
puntos < 7.5
```

Si la condición es cierta se entra en el cuerpo del bucle y se ejecutan sus sentencias. Una vez ejecutadas éstas se vuelve a la línea del *while* y se vuelve a evaluar la condición de nuevo. Si la condición es falsa no se entra al bucle y se continúa con las sentencias escritas debajo de *end loop*.

Cuando se escribe un bucle es muy importante pensar en lo siguiente:

1 **¿Se va a entrar al bucle la primera vez?** Esto depende del valor inicial de la condición.

2 **¿Qué queremos hacer en cada iteración?** (en cada ejecución del cuerpo del bucle).

3 **¿Cuándo se termina el bucle?** ¿Vamos a salir de el?

En el ejemplo, la primera vez entramos dado que *puntos* está inicializado a 0, lo que es menor que 7.5. En cada iteración vamos a leer una carta de la entrada estándar y a sumar su valor, acumulándolo en la variable *puntos*. Como las cartas siempre valen puntos tarde o temprano vamos a salir del bucle: en cuanto tengamos acumuladas en *puntos* las suficientes cartas como para que la condición sea *False*.

Si *puntos* es menor que 7.5 y leemos y sumamos otra carta que hace que *puntos* sea mayor o igual que 7.5, no leeremos ninguna carta más.

Comprobadas estas tres cosas parece que el bucle es correcto. Podríamos añadir a nuestro programa una condición para informar al usuario de si ha ganado el juego (tiene justo 7½ puntos) o si lo ha perdido (ha superado ese valor). Una vez terminado el bucle nunca podrá pasar que *puntos* sea menor, dado que en tal caso seguiríamos iterando.

```
puntos := 0.0;
while puntos < 7.5 loop
       LeerCarta(carta);
       puntos := puntos + ValorCarta(carta);
end loop;
if puntos = 7.5 then
       Put("Has ganado");
else
       Put("Has perdido");
end if;
New_Line;
```

7.2. Contar

El siguiente programa escribe los números del 1 al 5:

```
numero := 1;
while numero <= 5 loop
       Put(numero);
       numero := numero + 1;
end loop;
```

Como puede verse, inicialmente se inicializa *numero* a *1*. Esto hace que se entre al *while* dado que 1 es menor o igual que 5. Se imprime el número y se incrementa este. Y se sigue iterando o dando vueltas (de ahí la palabra reservada *loop*) en el *while* mientras la condición sea cierta. Cada vez que ejecutamos el cuerpo queremos imprimir el número actual y pasar al siguiente número. Se termina el bucle cuando el número pasa a valer 6, en cuyo caso acabamos de imprimir un 5 y no se vuelve a entrar en el bucle.

Es importante comprobar todo esto. Si el bucle hubiese sido

```
numero := 1;
while numero < 5 loop
       Put(numero);
       numero := numero + 1;
end loop;
```

no habríamos escrito el número 5, dado que incrementamos número al final del bucle y justo después comprobamos la condición (pero no imprimimos el número hasta entrar al bucle).

Si hubiésemos programado en cambio

```
numero := 1;
while numero <= 5 loop
       numero := numero + 1;
       Put(numero);
end loop;
```

entonces no habríamos escrito el número 1.

Comprobando los tres puntos (entrada, qué se hace en cada iteración y cómo se sale) no debería haber problema; los errores introducidos por accidente deberían verse rápidamente.

7.3. Repetir hasta que baste

Otro tipo de bucle es similar al anterior, pero comprobando la condición al final de cada ejecución del cuerpo y no al principio. Este bucle se suele conocer como *repeat-until* (o *repetir-hasta*) y lo podemos escribir en Ada como sigue:

```
loop
       sentencias;
       exit when condicion;          -- salir cuando...
end loop;
```

En este caso se ejecutan las sentencias del cuerpo del bucle en primer lugar. ¡Sin evaluar condición alguna! Por último se evalúa la condición. Si esta es cierta, se termina la ejecución del bucle y se continúa tras el. Si esta es falsa, se sigue de nuevo iterando.

Es también posible omitir la sentencia *exit when*, lo que crearía un bucle infinito. La figura 7.2 muestra el esquema del flujo de control en este tipo de bucle. (Ada permite escribir la sentencia *exit* en cualquier parte del cuerpo del bucle, pero en este curso obligamos a que se escriba como última sentencia del bucle, justo antes del *end loop*).

Por ejemplo, el siguiente bucle es más apropiado para escribir los números del uno al cinco, dado que sabemos que al menos vamos a escribir un número. Lo que es lo mismo, la primera vez siempre queremos entrar al bucle.

```
numero := 1;
loop
       Put(numero);
       numero := numero + 1;
       exit when numero = 6;
end loop;
```

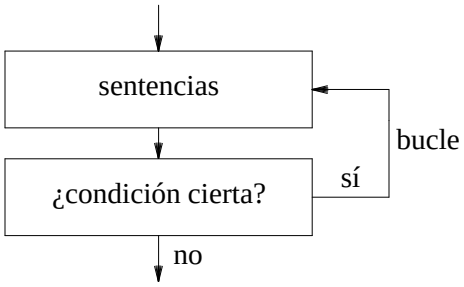


Figura 7.2: *Flujo de control en un bucle tipo repetir hasta o repeat-until.*

Como puede verse, también ha sido preciso inicializar *numero* con el primer valor que queremos imprimir. Si en esta inicialización hubiésemos dado a *numero* un valor mayor o igual a 6 entonces habríamos creado un bucle infinito.

Hay que tener siempre cuidado de comprobar las tres condiciones que hemos mencionado antes para los bucles. Es muy útil también comprobar qué sucede en la primera iteración y qué va a suceder en la última. De otro modo es muy fácil que iteremos una vez más, o menos, de las que precisamos debido a algún error. Si lo comprobamos justo cuando escribimos el bucle seguramente ahorremos mucho tiempo de depuración (que no suele ser un tiempo agradable).

7.4. Sabemos cuántas veces iterar

Los dos tipos de bucle que hemos visto iteran un número variable de veces; siempre dependiendo de la condición (el *while* comprobándola antes del cuerpo y el *loop* con *exit when* comprobándola después).

Hay un tercer tipo de bucle que itera un número determinado de veces. Este bucle es muy popular dado que en muchas ocasiones sí sabemos cuantas veces queremos iterar. Nos referimos al bucle **for**.

El bucle *for itera tantas veces como elementos tenga un rango*. Siempre. Este es su esquema:

```
for variable in primero..ultimo loop          -- para ... en ... repetir
       Sentencias;
end loop;
```

Este bucle itera tantas veces como elementos tiene el rango *primero..ultimo*. Para conseguirlo, cada bucle *for* utiliza una variable llamada **variable de control**, que toma valores sucesivos dentro del rango cada vez que se ejecuta el cuerpo del bucle. En la primera iteración la variable de control toma automáticamente el primer valor del rango. Se sigue repitiendo la iteración con sucesivos valores de dicho rango hasta agotarlo. Durante la última iteración la variable toma como valor el último elemento del rango.

En Ada, la variable de control no hay que declararla. Es el mismo *for* el que la declara; y tiene como ámbito el cuerpo del bucle, por lo que si queremos utilizar su valor pasado el bucle no podremos hacerlo (en otros lenguajes sí se permite). En tal caso es más recomendable utilizar un bucle *while* o utilizar una variable auxiliar. Por cierto, en muchos otros lenguajes esta variable sí hay que declararla y además sobrevive al bucle en que se utiliza, pero, como hemos dicho, este no es el caso en Ada.

Puede que no iteremos ninguna vez. En aquellas ocasiones en que *primero* es mayor que *ultimo* el bucle no itera ninguna vez, dado que en tal caso el rango se considera vacío. Esto puede suceder cuando ambos límites están determinados por variables o expresiones y no por literales.

Veamos un ejemplo. El siguiente bucle es realmente el que deberíamos escribir para imprimir los números del 1 al 5.

```
for i in 1..5 loop
      Put(i);
end loop;
```

Cuando vemos este bucle sabemos que el cuerpo ejecuta cinco veces. La primera vez *i* vale 1, la siguiente 2, luego 3, ... así hasta 5. Habría sido exactamente lo mismo ejecutar:

```
i := 1;
while i <= 5 loop
      Put(i);
      i := i + 1;
end loop;
```

Un bucle *for* se utiliza siempre que se conoce cuántas veces queremos ejecutar el cuerpo del bucle. En el resto de ocasiones lo normal es utilizar un *while*, salvo si sabemos que siempre vamos a entrar y que resulta cómodo comprobar la condición al final (en cuyo caso utilizamos un *repeat until*).

En muchas ocasiones la variable de control no se utiliza para nada en el programa. Para nada, salvo para conseguir que el bucle ejecute justo las veces que deseamos. Tal vez queramos hacer algo justo cinco veces, pero lo que hagamos sea siempre exactamente lo mismo, sin depender de cuál es la vez. En tal caso, para hacer algo 5 veces, escribiríamos un bucle *for* como el de arriba, pero sustituyendo la invocación a *Put* por las sentencias que queramos ejecutar 5 veces.

Esto se hace igual con todos los bucles. Una cosa son las sentencias que escribimos para controlar el bucle (cuándo entramos, cómo preparamos la siguiente pasada, cuándo salimos) y otra es lo que queremos hacer dentro del bucle. El bucle nos deja recorrer una serie de etapas. Lo que hagamos en cada etapa depende de las sentencias que incluyamos en el cuerpo del bucle.

Habrá ocasiones en que resulte útil recorrer un rango pero al revés, contando desde el último valor hasta el primero. Para esto puede escribirse la palabra reservada *reverse* antes del rango, como en este ejemplo:

```
for i in reverse 1..5 loop
      Put(i);
end loop;
```

Efectivamente, este bucle escribe los números del 5 al 1, inclusive ambos y en ese orden.

Una última nota respecto a estos bucles. Es tradicional utilizar nombres de variable tales como *i*, *j* y *k* para controlar los bucles. Eso sí, cuando la variable de control del bucle realmente represente algo en nuestro programa (más allá de cuál es el número de la iteración) será mucho mejor darle un nombre más adecuado; un nombre que indique lo que representa la variable.

7.5. Cuadrados

Queremos imprimir los cuadrados de los cinco primeros números positivos. Para hacer esto necesitamos repetir justo cinco veces las sentencias necesarias para elevar un número al cuadrado e imprimir su valor.

cuadrados.adb

```
1     --
2     -- Imprimir cuadrados de 1..5
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;
```

```
7     procedure Cuadrados is

9           MaxNum: constant Integer := 5;

11          package Integer_InOut is new Integer_IO(Integer);
12          use Integer_InOut;
13
14          function Cuadrado(n: Integer) return Integer is
15          begin
16               return n ** 2;
17          end;

19          x: Integer;
20    begin

22          for i in 1..MaxNum loop
23               x := Cuadrado(i);
24               Put(i);
25               Put(" ** 2 = ");
26               Put(x);
27               New_Line;
28          end loop;

30    end;
```

Dado que queremos imprimir justo los cuadrados de los cinco primeros números podemos utilizar un bucle *for* que itere justo en ese rango y utilizar la variable de control del bucle como número para elevar al cuadrado. El programa hace precisamente eso. Aunque no es realmente necesario, hemos utilizado una función para elevar el número al cuadrado.

Ahora hemos cambiado de opinión y deseamos imprimir los cuadrados menores o iguales a 100. Empezando por el 1 como el primer número a considerar. Esta vez no sabemos exactamente cuántas veces tendremos que iterar (podríamos hacer las cuentas pero es más fácil no hacerlo).

La estructura que determina el control sobre el bucle es el valor del cuadrado que vamos a imprimir. Hay que dejar de hacerlo cuando dicho cuadrado pase de 100. Luego podríamos escribir

```
n := 1;
n2 := 1;
while n2 <= 100 loop
      PutCuadrado(n, n2);
      n := n + 1;
      n2 := n ** 2;
end loop;
```

Aquí *n* es el número que elevamos al cuadrado. Inicialmente será 1 pero en cada pasada vamos a incrementarlo para elevar otro número más al cuadrado. Ahora bien, es *n2* (por n^2) la variable que determina que hay que seguir iterando. El programa completo podría quedar como sigue.

cuadrados.adb

```
1     --
2     -- Imprimir cuadrados menores o iguales que cien
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure Cuadrados is
```

```
9          package Integer_InOut is new Integer_IO(Integer);
10         use Integer_InOut;

12         procedure PutCuadrado(n: Integer; n2: Integer) is
13         begin
14              Put(n);
15              Put(" ** 2 = ");
16              Put(n2);
17              New_Line;
18         end;

20         n: Integer;
21         n2: Integer;
22    begin

24         n := 1;
25         n2 := 1;
26         while n2 <= 100 loop
27              PutCuadrado(n, n2);
28              n := n + 1;
29              n2 := n ** 2;
30         end loop;
31    end;
—
```

Podríamos haber escrito el bucle de este otro modo:

```
n := 1;
loop
       n2 := n ** 2;
       if n2 <= 100 then
              PutCuadrado(n, n2);
       end if;
       n := n + 1;
       exit when n2 > 100;
end loop;
```

Empezamos a contar en 1 y elevamos al cuadrado. Pasamos luego al siguiente número. Pero hemos de tener cuidado de no imprimir el cuadrado si hemos sobrepasado 100, por lo que es necesario un *if* que evite que se imprima un número cuando hemos pasado el límite.

Esta estructura, con un *if* para evitar que la acción suceda justo en la última iteración, es habitual si se utilizan bucles del estilo *repeat-until*. Debido a esto, en general, son más populares los bucles *while*.

7.6. Bucles anidados

Es posible anidar bucles (escribir uno dentro de otro) cuando queramos recorrer objetos que tengan varias dimensiones. Piensa que siempre es posible escribir sentencias dentro de sentencias compuestas tales como condicionales y bucles. Por ejemplo, el siguiente programa escribe los nombres de las casillas de un tablero de 10 x 10, donde la casilla (i, j) es la que está en la fila i y columna j.

```
for fila in 1..10 loop
       for columna in 1..10 loop
              Put("[");
              Put(i);
              Put(",");
              Put(j);
              Put("]");
              if columna < 10 then
                     Put(" ");
              end if;
       end loop;
       New_Line;
end loop;
```

El bucle externo (el que comienza en la primera línea) se ocupa de imprimir cada una de las filas. Su cuerpo hace que se imprima la fila *i*. El bucle interno se ocupa de imprimir las casillas de una fila. Su cuerpo se ocupa de imprimir la casilla *j*. Eso sí, su cuerpo hace uso del hecho de que estamos en la fila *i*. ¿Cuál crees que será la salida de este programa? ¿Por qué hay un *New_Line* tras el bucle más anidado?

7.7. Triángulos

Queremos dibujar un triángulo de altura dada como muestra la figura 7.3. Se trata de escribir empleando ‘‘*’’ una figura en la pantalla similar al triángulo de la figura.

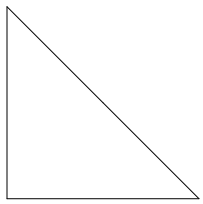


Figura 7.3: *Un triángulo rectángulo sobre un cateto.*

En este tipo de problemas es bueno dibujar primero (a mano) la salida del programa y pensar cómo hacerlo. Este es un ejemplo (hecho ejecutando el programa que mostramos luego):

```
; ada triangulo
; triangulo
*
**
***
****
*****
```

Parece que tenemos que dibujar tantas líneas hechas con ‘‘*’’ como altura nos dan (5 en este ejemplo). Por tanto el programa debería tener el aspecto

```
for i in 1..Alto loop
       ...
end loop;
```

Donde *Alto* es una constante que determina la altura.

Fijándonos ahora en cada línea, hemos de escribir tantos ‘‘*’’ como el número de línea en la que estamos. Luego el programa completo sería como sigue:

triangulo.adb

```
1     --
2     -- Dibujar un triangulo rectangulo sobre un cateto.
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure Triangulo is

9           Alto: constant Integer := 5;

11    begin
12          for i in 1..Alto loop
13                for j in 1..i loop
14                      Put("*");
15                end loop;
16                New_Line;
17          end loop;
18    end;
—
```

Ahora queremos dibujar dicho triángulo boca-abajo, como en la figura 7.4.

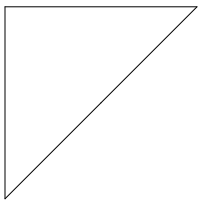


Figura 7.4: *Un triángulo invertido.*

Podríamos pensar de nuevo como hacerlo. No obstante, la única diferencia es que ahora las líneas están al revés. Luego podríamos cambiar el programa anterior para que utilice un bucle de cuenta hacia atrás:

```
for i in reverse 1..Alto loop
      for j in 1..i loop
            Put("*");
      end loop;
      New_Line;
end loop;
```

Nótese que utilizar *reverse* en el bucle interno no haría nada. Seguiríamos escribiendo una línea de la misma longitud.

El resultado de ejecutar este programa es como sigue:

```
; ada trianguloinv
; trianguloinv
*****
****
***
**
*
```

Podríamos también centrar el triángulo sobre la hipotenusa, como muestra la figura 7.5. Este problema requiere pensar un poco más.

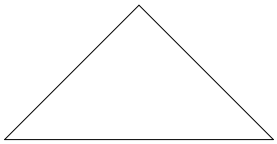


Figura 7.5: *Un triángulo sobre su hipotenusa.*

De nuevo, hay que escribir tantas líneas como altura tiene el triángulo (que es el valor dado). Luego el programa sigue teniendo el aspecto

```
for i in 1..Alto loop
       ...
end loop;
```

La diferencia es que ahora tenemos que escribir unos cuantos espacios en blanco al principio de cada línea, para que los ‘‘*’’ queden centrados formando un triángulo. Después hay que escribir los ‘‘*’’. Así pues el programa podría quedar con este aspecto:

```
for i in 1..Alto loop
       -- espacios a la izquierda.
       nblancos := ?????
       for j in 1..nblancos loop
              Put(" ");
       end loop;

       ancho := ????
       for j in 1..ancho loop
              Put("*");
       end loop;
       -- espacios a la derecha
       -- no hacen falta.
       New_Line;
end loop;
```

Si sabemos el número de espacios en cada línea y el ancho del triángulo en cada línea tenemos resuelto el problema.

¿Cuántos espacios hay que escribir en cada línea? ¡Fácil! Hay que escribir un triángulo invertido de espacios en blanco. La primera linea tiene más, la siguiente menos... así hasta la última que no tendría ninguno.

¿Cuántos ‘‘*’’? Bueno, el problema sólo especificaba la altura. Lo más sencillo es imprimir el doble del número de línea menos uno. (En la primera línea uno, en la siguiente tres, etc.). Por lo tanto el programa queda de este modo:

triangulo.adb

```
1     --
2     -- Dibujar un triangulo rectangulo sobre un cateto.
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure Triangulo is

9           Alto: constant Integer := 5;
```

```
11         nblancos: Integer;
12         ancho: Integer;
13    begin
14         for i in 1..Alto loop
15              -- espacios a la izquierda.
16              nblancos := alto - i;
17              for j in 1..nblancos loop
18                   Put(" ");
19              end loop;

21              -- relleno.
22              ancho := 2 * i  - 1;
23              for j in 1..ancho loop
24                   Put("*");
25              end loop;

27              -- espacios a la derecha
28              -- no hacen falta.
29              New_Line;
30         end loop;
31    end;
—
```

Ejecutar el programa produce la siguiente salida:

```
; ada triangulobase
; triangulobase
    *
   ***
  *****
 *******
*********
```

7.8. Primeros primos

Queremos imprimir los *n* primeros números primos. Lo que podemos hacer es ir recorriendo los números naturales e ir imprimiendo aquellos que son primos hasta tener *n*. El 1 es primo por definición, luego empezaremos a partir de ahí. Está todo hecho si pensamos en los problemas que hemos hecho, salvo ver si un número es primo o no. En tal caso, nos inventamos la función *EsPrimo* ahora mismo, lo que nos resuelve el problema. El programa tendría que tener el aspecto

```
num := 1;
llevamos := 0;
while llevamos < MaxPrimos loop
       if EsPrimo(num) then
              Put(num);
              llevamos := llevamos + 1;
       end if;
       num := num + 1;
end loop;
```

Iteramos mientras el número de primos que tenemos sea menor que el valor deseado. Por tanto necesitamos una variable para el número por el que vamos y otra para el número de primos que tenemos por el momento.

¿Cuándo es primo un número? Por definición, 1 lo es. Además si un número sólo es divisible por él mismo y por 1 también lo es. Luego en principio podríamos programar algo como

```
function EsPrimo(num: Integer) return boolean is
       loes: Boolean;
begin
       loes := True;
       for i in 2..num-1 loop
              if num mod i = 0 then
                     loes := False;
              end if;
       end loop;
       return loes;
end;
```

Inicialmente decimos que es primo. Y ahora, para todos los valores entre 2 y el anterior al número considerado, si encontramos uno que sea divisible entonces no lo es.

Lo que sucede es que es una pérdida de tiempo seguir comprobando números cuando sabemos que no es un primo. Por tanto, podemos utilizar un *while* que sea exactamente como el *for* que hemos utilizado pero... ¡Que no siga iterando si sabe que no es primo! Por lo demás, el programa está terminado y podría quedar como sigue:

primos.adb

```
1     --
2     -- Imprimir n numeros primos
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure Primos is

9           MaxPrimos: constant Integer := 15;

11          package Int_IO is new Integer_IO(Integer);
12          use Int_IO;

14          function EsPrimo(num: Integer) return boolean is
15                loes: Boolean;
16                i: Integer;
17          begin
18                loes := True;
19                i := 2;
20                while i < num and loes loop
21                      loes := num mod i /= 0;
22                      i := i + 1;
23                end loop;
24                return loes;
25          end;
```

```
27          num: Integer;
28          llevamos: Integer;
29     begin
30          num := 1;
31          llevamos := 0;
32          while llevamos < MaxPrimos loop
33               if EsPrimo(num) then
34                    Put(num);
35                    llevamos := llevamos + 1;
36               end if;
37               num := num + 1;
38          end loop;
39     end;
—
```

La salida del programa queda un poco fea, puesto que se imprimen todos los primos en una única línea. Imprimirlos uno por línea puede quedar feo por utilizar tantas líneas. Podemos mejorar un poco el programa haciendo que cada cinco números primos se salte a la siguiente línea. Esto lo podemos hacer si modificamos el cuerpo del programa principal como sigue:

```
num := 1;
llevamos := 0;
while llevamos < MaxPrimos loop
       if EsPrimo(num) then
              Put(num);
              llevamos := llevamos + 1;
              if llevamos mod 5 = 0 then
                     New_Line;
              end if;
       end if;
       num := num + 1;
end loop;
```

La idea es que cuando *llevamos* sea 5 queremos saltar de línea. Cuando sea 10 también. Cuando sea 15 también... Luego queremos que cuando *llevamos* sea múltiplo de 5 se produzca un salto de línea. Nótese que si este nuevo *if* lo ponemos fuera del otro *if* entonces saltaremos muchas veces de línea ¿Ves por qué?.

7.9. ¿Cuánto tardará mi programa?

Este último programa tiene dos bucles anidados. Por un lado estamos recorriendo números y por otro en cada iteración llamamos a *EsPrimo* que también recorre los números (hasta el que llevamos). Utilizar funciones para simplificar las cosas puede tal vez ocultar los bucles, pero hay que ser consciente de que están ahí.

En general, cuánto va tardar un programa en ejecutar es algo que no se puede saber. Como curiosidad, diremos también que de hecho resulta imposible saber automáticamente si un programa va a terminar de ejecutar o no. Pero volviendo al tiempo que requiere un programa... ¡Depende de los datos de entrada! (además de depender del algoritmo). Pero sí podemos tener una idea aproximada.

Tener una idea aproximada es importante. Nos puede ayudar a ver cómo podemos mejorar el programa para que tarde menos. Por ejemplo, hace poco cambiamos un *for* por un *while* precisamente para no recorrer todos los números de un rango innecesariamente. En lo que a nosotros se refiere, vamos a conformarnos con las siguientes ideas respecto a cuanto tardará mi programa:

- Cualquier sentencia elemental supondremos que tarda 1 en ejecutar. (Nos da igual si es un nanosegundo o cualquier otra unidad; sólo queremos ver si un programa va a tardar más o menos que otro). Pero... ¡Cuidado!, una sentencia que asigna un *record* de 15 campos tarda

15 veces más que asignar un entero a otro.

- Un bucle que recorre *n* elementos va a tardar *n* unidades en ejecutar.
- Un bucle que recorre *n* elementos, pero que deja de iterar cuando encuentra un elemento, suponemos que tarda *n*/2 en ejecutar.

Así pues, dos bucles *for* anidados suponemos que en general tardan n^2. Tres bucles anidados tardarían n^3. Y así sucesivamente.

Aunque a nosotros nos bastan estos rudimentos, es importante aprender a ver cuánto tardarán los programas de un modo más preciso. A dichas técnicas se las conoce como el estudio de la **complejidad** de los algoritmos. Cualquier libro de algorítmica básica contiene una descripción razonable sobre cómo estimar la complejidad de un algoritmo. Aquí sólo estamos aprendiendo a programar de forma básica y esto nos basta:

1 Lo más importante es que el programa sea correcto y se entienda bien.

2 Lo siguiente más importante es que acabe cuanto antes. Esto es, que sea lo más eficiente posible en cuanto al tiempo que necesita para ejecutar.

3 Lo siguiente más importante es que no consuma memoria de forma innecesaria.

Problemas

Recuerda que cuando encuentres un enunciado cuya solución ya has visto queremos que intentes hacerlo de nuevo sin mirar la solución en absoluto.

1 Calcular un número elevado a una potencia sin utilizar el operador de exponenciación de Ada.

2 Escribir las tablas de multiplicar hasta el 11.

3 Calcular el factorial de un número dado.

4 Calcular un número combinatorio. Expresarlo como un tipo de datos e implementar una función que devuelva su valor.

5 Leer números de la entrada y sumarlos hasta que la suma sea cero.

6 Leer cartas de la entrada estándar y sumar su valor (según el juego de las 7 y media) hasta que la suma sea 7.5. Si la suma excede 7.5 el juego ha de comenzar de nuevo (tras imprimir un mensaje que avise al usuario de que ha perdido la mano).

7 Dibujar un rectángulo sólido con asteriscos dada la base y la altura.

8 Dibujar un rectángulo hueco con asteriscos dada la base y la altura.

9 Dibujar un tablero de ajedrez utilizando espacios en blanco para las casillas blancas, el carácter ‘‘X’’ para las casillas negras, barras verticales y guiones para los bordes y signos ‘‘+’’ para las esquinas.

10 Escribir un triángulo sólido en la salida estándar dibujado con asteriscos, dada la altura. El triángulo es rectángulo y está apoyado sobre su cateto menor con su cateto mayor situado a la izquierda.

11 Escribir en la salida estándar un triángulo similar al anterior pero cabeza abajo (con la base arriba).

12 Escribir un triángulo similar al anterior pero apoyado sobre la hipotenusa.

13 La serie de Fibonacci se define suponiendo que los dos primeros términos son 0 y 1. Cada nuevo término es la suma de los dos anteriores. Imprimir los cien primeros números de la serie de fibonacci.

14 Calcular los 10 primeros números primos.

15 Leer desde la entrada estándar fechas hasta que la fecha escrita sea una fecha válida.

16 Calcular el número de días entre dos fechas teniendo en cuenta años bisiestos.

17 Imprimir los dígitos de un número dado en base decimal.

18 Leer números de la entrada estándar e imprimir el máximo, el mínimo y la media de todos ellos.

19 Buscar el algoritmo de Euclides para el máximo común divisor de dos números e implementarlo en Ada.

20 Escribir los factores de un número.

21 Dibuja la gráfica de una función expresada en Ada empleando asteriscos.

22 Algunos de los problemas anteriores de este curso han utilizado múltiples sentencias en secuencia cuando en realidad deberían haber utilizado bucles. Localízalos y arréglalos.

8 — Colecciones de elementos

8.1. Arrays

Las tuplas son muy útiles para modelar elementos de mundos abstractos, o de nuevos tipos de datos. No obstante, en muchas ocasiones tenemos objetos formados por una secuencia ordenada de elementos del mismo tipo. Por ejemplo: un punto en dos dimensiones son dos números; una baraja de cartas es una colección de cartas; un mensaje de texto es una secuencia de caracteres; una serie numérica es una colección de números; y podríamos seguir con innumerables ejemplos.

Un **array** es una colección ordenada de elementos del mismo tipo que tiene como propiedad que a cada elemento le corresponde un índice (un número natural). A este tipo de objeto se le denomina **vector** o bien **colección indexada** de elementos. Aunque se suele utilizar el término *array* para referirse a ella.

Un vector de los utilizados en matemáticas es un buen ejemplo del mismo concepto. Por ejemplo, el vector $\vec{a} = (a_1, a_2, a_3)$ es un objeto compuesto de tres elementos del mismo tipo: a_1, a_2 y a_3. Además, a cada elemento le corresponde un número o índice (1 al primero, 2 al segundo y 3 al tercero). El concepto de *array* es similar y permite que, dado un índice, podamos recuperar un elemento de la colección de forma inmediata.

Un *array* en Ada puede declararse empleando la palabra reservada *array*, tal y como sigue:

```
type TipoArray is array(indice1..indiceN) of TipoElemento;
```

Donde *indice1..indiceN* ha de ser un rango para los índices de los elementos del *array* y *TipoElemento* es el tipo de datos para los elementos del *array*. El rango puede ser cualquier rango de cualquier tipo enumerado. En la mayoría de los casos se utilizan enteros utilizando 1 como primer elemento del rango (en otros lenguajes la costumbre es comenzar en 0, pero no así en Ada).

Por ejemplo, si consideramos las notas de un alumno en 4 semanas de curso tendremos un buen ejemplo de un objeto abstracto (‘‘notas de un curso’’) que puede representarse fácilmente con un *array*. Tenemos 4 números que constituyen las notas de un alumno. Cada número tiene asignada una posición de 1 a 4 dado que se corresponde con una semana dada. Estas notas, conjuntamente, serían un vector de notas que podemos definir en Ada como sigue:

```
type TipoNotas is array(1..4) of Float;
```

Esto declara el nuevo tipo de datos *TipoNotas* como un *array* de elementos de tipo *Float* (de reales) que tiene cuatro elementos con índices 1, 2, 3 y 4. Una vez declarado el tipo podemos declarar variables de dicho tipo, como por ejemplo:

```
notas: TipoNotas;
```

Habitualmente declararíamos un subtipo para el rango empleado para los índices antes de declarar el tipo para el *array*. Además se suele declarar una constante para indicar el tamaño del *array*. Por ejemplo, habría sido más correcto declarar:

```
MaxNota: constant Integer := 4;

subtype TipoRangoNotas is Integer range 1..MaxNota;

type TipoNotas is array(TipoRangoNotas) of Float;
```

En la memoria del ordenador el *array* puede imaginarse como una secuencia de los elementos del *array*, uno a continuación de otro, tal y como muestra la figura 8.1.

Podemos tener *arrays* de muy diversos tipos. Como tipo de datos para el elemento puede utilizarse cualquier tipo de datos. Como tipo de datos para el índice puede utilizarse cualquier rango de un tipo enumerado (por ejemplo, caracteres, días de la semana, etc.).

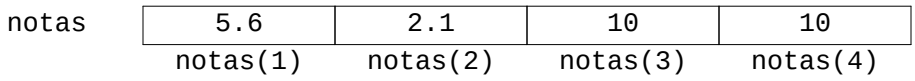


Figura 8.1: *Aspecto de un array en la memoria del ordenador.*

El siguiente tipo pretende describir cómo nos ha ido cada día de la semana:

```
type TipoDiaSem is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
type TipoQueTal is (MuyMal, Mal, Regular, Bien, MuyBien);
type TipoQueTalSem is array(Lun..Dom) of TipoQueTal;
```

Declarar una variable de este tipo se hace como de costumbre:

```
misemana: TipoQueTalSem;
```

El tipo *TipoQueTalSem* define un *array* de elementos de tipo *TipoQueTal* de tal forma que para cada día de la semana (para cada índice) podemos guardar que tal nos ha ido ese día. Este tipo es similar al *array* de notas mostrado antes. La única diferencia es que ahora los índices van de *Lun* a *Dom* en lugar de ir de 1 a 4 y que los elementos del *array* no son números reales, sino elementos de *TipoQueTal*.

Dado un índice es inmediato obtener el elemento correspondiente del *array*. Este tipo de acceso supone más o menos el mismo tiempo que acceder a una variable cualquiera (y lo mismo sucede al acceder a los campos de un *record*). Para eso sirven los *arrays*. Así, si en el ejemplo anterior consideramos el índice *Mie*, podemos directamente encontrar el elemento para ese día en el *array*. En matemáticas se suelen utilizar subíndices para expresar los índices de los *arrays*. En Ada escribimos el índice entre paréntesis tras el nombre de la variable de tipo *array*. Por ejemplo, la siguiente expresión corresponde a que tal nos ha ido el miércoles:

```
misemana(Mie)
```

Esta expresión es en realidad una variable. Esto es, la podemos escribir en la parte izquierda o derecha de una asignación (lo mismo que sucedía con los campos de un *record*). Por ejemplo, esto hace que oficialmente el miércoles haya ido muy bien:

```
misemana(Mie) := MuyBien;
```

Y esto hace que el domingo nos haya ido como nos ha ido el miércoles:

```
misemana(Dom) := misemana(Mie);
```

Los bucles resultan particularmente útiles para trabajar sobre *arrays*, dado que los índices son elementos enumerados de valores consecutivos. Este fragmento de código imprime por la salida que tal nos ha ido durante la semana:

```
for dia in Lun..Dom loop
      Put(misemana(dia));
end loop;
```

Y cuidado aquí. *Put* es un subprograma y los paréntesis se utilizan para representar una llamada. En cambio, *misemana* es un *array* y los paréntesis se utilizan para representar una indexación o acceso mediante índice al *array*.

Los tipos array en Ada poseen el **atributo Range** que permite conocer el rango del *array* dado el tipo. Así, *TipoNotas'Range* es equivalente a *1..4* y `TipoQueTalSem'Range` es equivalente a *Lun..Dom*. También tenemos el atributo *Length*, que está definido como el número

de elementos en el *array*. Podemos utilizar los atributos para escribir el bucle anterior un poco mejor:

```
for dia in TipoQueTalSem'Range loop
      Put(misemana(dia));
end loop;
```

Lo bueno de escribir el bucle así es que sabemos que la variable *dia* va a iterar sobre los elementos utilizados como rango para los índices de *TipoQueTalSem*. Si ahora cambiamos la declaración de *TipoQueTalSem* para que utilice otro rango como índice no es preciso cambiar el bucle. El bucle sigue funcionando.

Existen otros dos atributos muy útiles al manipular *arrays*: *First* corresponde al primer índice y *Last* corresponde al último. Por ejemplo, el siguiente bucle es exactamente igual a los dos anteriores:

```
for dia in TipoQueTalSem'First..TipoQueTalSem'Last loop
      Put(misemana(dia));
end loop;
```

Debería resultar obvio que utilizar un índice fuera de rango supone un error. Por ejemplo, dadas las declaraciones para notas utilizadas anteriormente como ejemplo, utilizar la siguiente secuencia de sentencias provocará un error que detendrá la ejecución del programa dado que en el *array* de notas definido el 7 no es un índice válido.

```
valor := 7;
notas(valor) := 3.5;          -- Error: indice fuera de rango.
```

Para inicializar constantes de tipo *array* es útil utilizar agregados (de forma similar a cuando los utilizamos para inicializar *records*). Este código define un vector en el espacio discreto de tres dimensiones y define la constante *Origen* como el vector cuyos tres elementos son cero. Esta inicialización es equivalente a asignar sucesivamente los valores del agregado a las posiciones del *array*:

```
type TipoVector is array(1..3) of Integer;

Origen: constant TipoVector := (0, 0, 0);
```

Se permite asignar *arrays* del mismo tipo entre sí. Naturalmente, dichos *arrays* han de ser del mismo tamaño y los elementos han de ser del mismo tipo. También se permite comparar *arrays* del mismo tipo entre sí, pero sólo comparaciones de igualdad o desigualdad. Estas operaciones funcionan elemento a elemento: la asignación copia elemento por elemento y la comparación considera que dos *arrays* son iguales si son iguales elemento a elemento.

Otro concepto que puede resultar muy útil a la hora de manipular *arrays* es el concepto de **rebanada**. Una rebanada es un trozo de un *array* delimitado por un rango. Por ejemplo, *letras(2..4)* es el *array* formado por los elementos con índices 2, 3 y 4 de *letras*. Ada permite asignar rebanadas entre sí, de tal forma que

```
for i in 3..6 loop
      destino(i) := origen(i);
end loop
```

podría también escribirse como

```
destino(3..6) := origen(3..6);
```

8.2. Problemas de colecciones

Sólo hay tres tipos de problemas en este mundo:

1 Problemas de solución directa. Los primeros que vimos.

2 Problemas de casos. Lo siguientes que vimos. Tenías que ver qué casos tenías y resolverlos por separado.

3 Problemas de colecciones de datos.

Los problemas que encontrarás que requieren recorrer colecciones de datos van a ser variantes o combinaciones de los problemas que vamos a resolver a continuación. Concretamente van a consistir todos en:

- acumular un valor, o
- buscar un valor, o
- maximizar (o minimizar) un valor, o
- construir algo a partir de los datos.

Si dominas estos problemas los dominas todos

Por eso es muy importante que entiendas bien los ejemplos que siguen y juegues con ellos. Justo a continuación vamos a ver ejemplos de problemas de acumulación, búsqueda y maximización. Veremos también problemas de construcción pero lo haremos después como ejemplos de uso de cadenas de caracteres y operaciones en ficheros.

8.3. Acumulación de estadísticas

Tenemos unos enteros, procedentes de unas medidas estadísticas, y queremos obtener otras ciertas estadísticas a partir de ellos. Concretamente, estamos interesados en obtener la suma de todas las medidas y la media.

Podemos empezar por definir nuestro tipo de datos de estadísticas y luego nos preocuparemos de programar por separado un subprograma para cada resultado de interés.

Nuestras estadísticas son un número concreto de enteros, en secuencia. Esto es un *array*.

```
1     MaxNum: constant Integer := 5;

3     type TipoEstadisticas is array(1..MaxNum) of Integer;
```

Utilizamos una constante *MaxNum* para poder cambiar fácilmente el tamaño de nuestra colección de estadísticas. De hecho, podríamos haber definido un subtipo rango para los índices.

Sumar las estadísticas requiere **acumular** en una variable la suma de cada uno de los valores del *array*. Esta función hace justo eso.

```
1     function Suma(estadisticas: TipoEstadisticas) return Integer is
2          suma: Integer;
3     begin
4          suma := 0;
5          for i in TipoEstadisticas'Range loop
6               suma := suma + estadisticas(i);
7          end loop;
8          return suma;
9     end;
```

Esta forma de manipular un *array* es muy típica. Partimos de un valor inicial (0 para la *suma*) y luego recorremos el *array* con un bucle *for*. El bucle itera sobre el rango de los índices (hemos

utilizado el atributo *Range* pero podríamos haber definido un tipo para el rango y haber utilizado dicho tipo). En cada iteración el bucle acumula (en *suma*) el valor en la posición *i* del *array*. El valor que tenemos tras el bucle es el resultado buscado.

Para calcular la media basta dividir la suma de todos los valores por el número de elementos (nos habríamos inventado *suma* si sólo nos hubiesen pedido hacer *Media*). El programa que sigue incluye la función *Media* y además ejercita el código que hemos escrito haciendo varias pruebas.

estadisticas.adb

```
1     --
2     -- Imprimir estadisticas
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure Estadisticas is

10          MaxNum: constant Integer := 5;

12          type TipoEstadisticas is array(1..MaxNum) of Integer;

14          package Integer_InOut is new Integer_IO(Integer);
15          use Integer_InOut;

17          function Suma(estadisticas: TipoEstadisticas) return Integer is
18               suma: Integer;
19          begin
20               suma := 0;
21               for i in TipoEstadisticas'Range loop
22                    suma := suma + estadisticas(i);
23               end loop;
24               return suma;
25          end;

27          function Media(estadisticas: TipoEstadisticas) return Integer is
28          begin
29               return Suma(estadisticas) / (estadisticas'Last - estadisticas'First + 1);
30          end;

32    --
33    -- Constantes de Prueba
34    --
35          Estadistica1: constant TipoEstadisticas := (1, 45, 3, 2, 4);
36          Prueba1: constant Integer := Suma(Estadistica1);
37          Prueba2: constant Integer := Media(Estadistica1);

39    begin
40          Put(Prueba1);
41          New_Line;

43          Put(Prueba2);
44          New_Line;
45    end;
```

¿A ti también te sale 55 y 11 como resultado?

8.4. Buscar ceros

Sospechamos que alguno de nuestros valores estadísticos es un cero y queremos saber si es así y dónde está el cero si lo hay.

Necesitamos un procedimiento que nos diga si ha encontrado un cero y nos informe de la posición en que está (si está). Si tenemos dicho procedimiento podemos hacer nuestro programa llamándolo como en este programa:

```
1     BuscarCero(arraydenumeros, pos, encontrado);
2     if encontrado then
3          Put("pos = ");
4          Put(pos);
5     else
6          Put("no hay ninguno");
7     end if;
8     New_Line;
```

La clave del procedimiento *BuscarCero* es que debe utilizar un *while* para dejar de buscar si lo ha encontrado ya. El bucle es prácticamente un bucle *for*, pero teniendo cuidado de no seguir iterando si una variable *encontrado* informa de que hemos encontrado el valor buscado. Además, esa variable es uno de los dos parámetros que tenemos que devolver. El otro es la posición en que hemos encontrado el valor.

```
1     -- Busca un cero e informa de su posicion si se ha encontrado.
2     procedure BuscarCero(nums: TipoNums; i: out Integer; encontrado: out Boolean) is
3     begin
4          encontrado := False;
5          i := 1;
6          while i <= TipoNums'Last and not encontrado loop
7               if nums(i) = 0 then
8                    encontrado := True;
9               else
10                   i := i + 1;
11              end if;
12         end loop;
13    end;
```

Fíjate en que si el número es el que buscamos entonces no incrementamos *i*.

Es muy común tener que programar procedimientos similares a este. Siempre que se busque un valor en una colección de elementos el esquema es el mismo.

Una variante del problema anterior es ver si todos los valores son cero. En este caso lo único que nos interesa como resultado de nuestro subprograma es si todos son cero o no lo son. De nuevo se utiliza un *while*, para dejar de seguir buscando en cuanto sepamos que no todos son cero. Mostramos ahora el programa completo, con pruebas, en lugar de sólo el procedimiento.

todoscero.adb

```
1     --
2     -- Averigua si todos los numeros son cero en un array
3     --

5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure TodosCero is

10         MaxNum: constant Integer := 5;

12         type TipoNums is array(1..MaxNum) of Integer;
```

```
14          package Bool_IO is new Enumeration_IO(Boolean);
15          use Bool_IO;

17          -- Cierto si son todos cero.
18          function TodosCero(nums: TipoNums) return Boolean is
19               soncero: Boolean;
20               i: Integer;
21          begin
22               soncero := True;
23               i := 1;
24               while i <= TipoNums'Last and soncero loop
25                    if nums(i) /= 0 then
26                         soncero := False;
27                    else
28                         i := i + 1;
29                    end if;
30               end loop;
31               return soncero;
32          end;

34          -- Constantes de prueba
35          Prueba1: TipoNums := (43, 2, 45, 0, 43);
36          Prueba2: TipoNums := (0, 0, 0, 0, 0);
37     begin
38          Put(TodosCero(Prueba1));
39          New_Line;
40          Put(TodosCero(Prueba2));
41          New_Line;
42     end;
—
```

El *while* dentro de la función *TodosCero* es el que se ocupa de buscar dentro de nuestro *array*; esta vez está buscando algún elemento que no sea cero. Como verás, el esquema es prácticamente el mismo que cuando buscábamos un único elemento a cero en el *array*.

Aunque estamos utilizando *arrays* de enteros, debería quedar claro que podemos aplicar estas técnicas a cualquier tipo de *array*. Por ejemplo, podríamos buscar si nos ha ido muy mal algún día de la semana, en lugar de buscar un cero.

8.5. Buscar los extremos

Este problema consiste en buscar los valores máximo y mínimo de un *array* de números.

Para buscar el máximo se toma el primer elemento como candidato a máximo y luego se recorre *toda* la colección de números. Por tanto utilizamos un *for* esta vez. En cada iteración hay que comprobar si el nuevo número es mayor que nuestro candidato a máximo. En tal caso tenemos que cambiar nuestro candidato.

Este procedimiento es un ejemplo. Devuelve tanto el máximo como el mínimo.

```
1     procedure Extremos(nums: TipoNums; min: out Integer; max: out Integer) is
2     begin
3          min := nums(nums'First);
4          max := nums(nums'First);
```

```
5          for i in TipoNums'Range loop
6               if nums(i) < min then
7                    min := nums(i);
8               end if;
9               if nums(i) > max then
10                   max := nums(i);
11              end if;
12         end loop;
13    end;
```

Sería fácil modificarlo para que además devolviese los índices en que se hayan el máximo y el mínimo. Bastaría considerar una posición candidata a máximo además de un valor candidato a máximo (lo mismo para el mínimo). En cada iteración tendríamos que actualizar esta posición para el máximo cada vez que actualizásemos el valor máximo (y lo mismo para el mínimo). Inténtalo tú.

8.6. Ordenación

Queremos ordenar una secuencia de números de menor a mayor. Suponemos que tenemos declarado un tipo para un *array* de números llamado *TipoNums* y queremos ordenar un *array nums* de tipo *TipoNums*.

Hay muchas formas de ordenar un *array*, pero tal vez lo más sencillo sea pensar cómo lo ordenaríamos nosotros. Una forma es tomar el menor elemento del *array* y situarlo en la primera posición. Después, tomar el menor elemento de entre los que faltan por ordenar y moverlo a la segunda posición. Si seguimos así hasta el final habremos ordenado el *array*.

Siendo así tenemos que programar un bucle que haga esto. Para hacerlo es útil pensar en qué es lo que tiene que hacer el bucle en general, esto es, en la pasada i-ésima. Piensa que el código ha de funcionar sea cual sea la iteración en la que estamos.

La figura 8.2 muestra la idea para este problema. En la pasada número *i* tendremos ordenados los elementos hasta la posición anterior a la *i* en el *array* y nos faltarán por ordenar los elementos desde el i-ésimo hasta el último. Vamos a empezar por el elemento uno y, en este momento, no hay ninguna parte del *array* que esté ordenada. En esta situación tomamos el menor elemento del resto del *array* para dejarlo justo en la posición *i*.

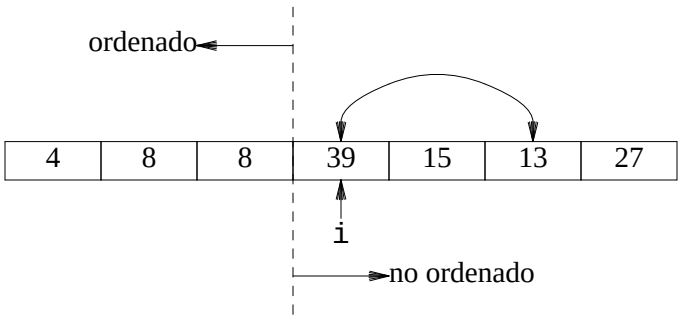


Figura 8.2: *Esquema para ordenar un array.*

Podríamos empezar por programar algo como:

```
1     for i in nums'First..nums'Last loop
2          -- buscar el menor de los que faltan
3          -- dejarlo en la posicion i
4     end loop
```

La pregunta es: ¿Cuáles son los que faltan por ordenar? Bueno, la idea de este bucle es que todos los que son menores que *i* (inicialmente ninguno) ya están ordenados. Los que faltan son todos

los números desde la posición *i* hasta el fin del *array*. Podríamos entonces escribir algo como esto:

```
1     for i in nums'First..nums'Last loop
2          BuscarMenor(nums(i..nums'Last), min, minpos);
3          Intercambiar(nums(i), nums(minpos));
4     end loop;
```

Utilizamos un procedimiento (que nos inventamos) *BuscarMenor* para localizar la posición del menor número de un *array* y le pasamos el trozo del *array* que nos falta por ordenar. Podríamos haber escrito otro bucle *for* anidado para localizar el menor elemento desde *nums(i)* hasta *nums(nums'Last)* pero parece más sencillo usar otro subprograma para esto. Por cierto, como intentamos hacer subprogramas de propósito general (que sirvan también en general) el procedimiento *BuscarMenor* va a devolver también el menor valor y no sólo su posición.

Una vez hemos encontrado el menor tenemos que intercambiarlo por *nums(i)*. Si no hacemos esto y simplemente asignamos el menor a *nums(i)* entonces perderemos el valor que hay en *nums(i)*. Y no queremos perderlo.

Aunque la idea está bien podemos afinar un poco más. No es preciso realizar la última iteración, dado que un *array* con un sólo elemento ya está ordenado. Además, tampoco es preciso intercambiar *nums(i)* y *nums(minpos)* si resulta que *i* es igual que *minpos*; esto puede pasar si el menor elemento de los que restan por considerar es justo el que estamos mirando. Considerando lo dicho, podemos escribir:

```
1     for i in nums'First..nums'Last-1 loop
2          BuscarMenor(nums(i..nums'Last), min, minpos);
3          if minpos /= i then
4               Intercambiar(nums(i), nums(minpos));
5          end if;
6     end loop;
```

El programa completo podría quedar tal y como se ve a continuación.

ordenar.adb

```
1     --
2     -- Ordenar una secuencia de numeros
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure Ordenar is

9          package Integer_InOut is new Integer_IO(Integer);
10         use Integer_InOut;

12         -- Para poder trabajar con arrays de cualquier tamaño.
13         type TipoNums is array(Integer range <>) of Integer;

15         procedure BuscarMenor(nums: TipoNums; min: out Integer; minpos: out Integer) is
16         begin
17              minpos := nums'First;
18              min := nums(minpos);
19              for i in nums'First+1..nums'Last loop
20                   if nums(i) < min then
21                        min := nums(i);
22                        minpos := i;
23                   end if;
24              end loop;
25         end;
```

```
27         procedure Intercambiar(n1: in out Integer; n2: in out Integer) is
28              aux: Integer;
29         begin
30              aux := n1;
31              n1 := n2;
32              n2 := aux;
33         end;

35         procedure OrdenarNums(nums: in out TipoNums) is
36              min: Integer;
37              minpos: Integer;
38         begin
39              for i in nums'First..nums'Last-1 loop
40                   BuscarMenor(nums(i..nums'Last), min, minpos);
41                   if minpos /= i then
42                        Intercambiar(nums(i), nums(minpos));
43                   end if;
44              end loop;
45         end;

47         procedure EscribirNums(nums: in TipoNums) is
48         begin
49              for i in nums'Range loop
50                   Put(nums(i));
51              end loop;
52              New_Line;
53         end;

55         -- Constantes de Prueba
56         Prueba1: TipoNums := (1, 2, 3, 4, 5);
57         Prueba2: TipoNums := (1, 1, 1);
58         Prueba3: TipoNums := (3, 543, 23, 0, 0, 2, 2, 3);

60    begin
61         -- Prueba
62         OrdenarNums(Prueba1);
63         EscribirNums(Prueba1);
64         OrdenarNums(Prueba2);
65         EscribirNums(Prueba2);
66         OrdenarNums(Prueba3);
67         EscribirNums(Prueba3);
68    end;
—
```

8.7. Búsqueda en secuencias ordenadas

Con el problema anterior sabemos cómo ordenar secuencias. ¿Y si ahora queremos buscar un número en una secuencia ordenada? Podemos aplicar la misma técnica que cuando hemos buscado un cero en un ejercicio previo. No obstante, podemos ser mas astutos: si en algún momento encontramos un valor mayor que el que buscamos, entonces el número no está en nuestra colección de números y podemos dejar de buscar. De no ser así la secuencia no estaría ordenada.

Este procedimiento busca el valor que se le indica en la colección de números (ordenada) que también se le indica.

```
1     procedure BuscarNum(nums: in TipoNums; valor: in Integer;
2                         pos: out Integer; encontrado: out Boolean) is
3          i: Integer;
4          puedeestar: Boolean;
5     begin
6          i := nums'First;
7          encontrado := False;
8          puedeestar := True;
9          while i <= nums'Last and not encontrado and puedeestar loop
10              encontrado := nums(i) = valor;
11              if encontrado then
12                   pos := i;
13              else
14                   puedeestar := nums(i) < valor;
15                   i := i + 1;
16              end if;
17         end loop;
18    end;
```

Su código es similar al que ya vimos para buscar ceros; pero fíjate en *puedeestar*. Hemos incluido otra condición más para seguir buscando: que pueda estar. Sabemos que si *nums(i)* es mayor que *valor* entonces no es posible que *valor* esté en el *array*, supuesto que este está ordenado.

¿Y no podemos hacerlo mejor? Desde luego que sí. El procedimiento anterior recorre en general todo el *array* (digamos que tarda $n/2$ en media). Pero si aplicamos lo que haríamos nosotros al buscar en un diccionario entonces podemos conseguir un procedimiento que tarde sólo $\log_2 n$ (¡Mucho menos tiempo! Cuando n es 1000, $n/2$ es 500 pero $\log_2 n$ es 10).

La idea es mirar a la mitad del *array*. Si el número que tenemos allí es menor que el que buscamos entonces podemos ignorar toda la primera mitad del *array*. ¡Hemos conseguido de un plumazo dividir el tamaño de nuestro problema a la mitad! Volvemos a aplicar la misma idea de nuevo. Tomamos el elemento en la mitad (de la mitad que ya teníamos). Si ahora ese número es mayor que el que buscamos entonces sólo nos interesa buscar en la primera mitad de nuestro nuevo conjunto de números. Y así hasta que o bien el número que miramos (en la mitad) sea el que buscamos o bien no tengamos números en que buscar.

A este procedimiento se le denomina **búsqueda binaria** y es un procedimiento muy popular. Por cierto, esta misma idea se puede aplicar a otros muchos problemas para reducir el tiempo que tardan en resolverse. Aunque no tienes por qué preocuparte ahora de este tema. Ya tenemos bastante con aprender a programar y con hacerlo bien.

```
1     procedure BusquedaBin(nums: in TipoNums; valor: in Integer;
2                         pos: out Integer; encontrado: out Boolean) is
3          izq: Integer;
4          der: Integer;
5          med: Integer;
6          puedeestar: Boolean;
7     begin
8          izq := nums'First;
9          der := nums'Last;
10         encontrado := False;
```

```
12          while not encontrado and izq <= der loop
13               med := (izq + der) / 2;
14               if nums(med) = valor then
15                    encontrado := True;
16                    pos := med;
17               elsif nums(med) < valor then
18                    izq := med + 1;
19               else
20                    der := med - 1;
21               end if;
22          end loop;
23     end;
```

En cada iteración del *while* estamos en la situación que muestra la figura 8.3. El valor que buscamos sólo puede estar entre los elementos que se encuentran en las posiciones del rango *izq..der*. Inicialmente consideramos todo el *array*. Así que nos fijamos en el valor que hay en la posición media: *nums(med)*. Si es el que buscamos ya está todo hecho. En otro caso o bien cambiamos nuestra posición *izq* o nuestra posición *der* para ignorar la mitad del *array* que sabemos es irrelevante para la búsqueda. Cuando ignoramos una mitad del *array* también ignoramos la posición que antes era la mitad (por eso se suma o se resta uno en el código que hemos mostrado).

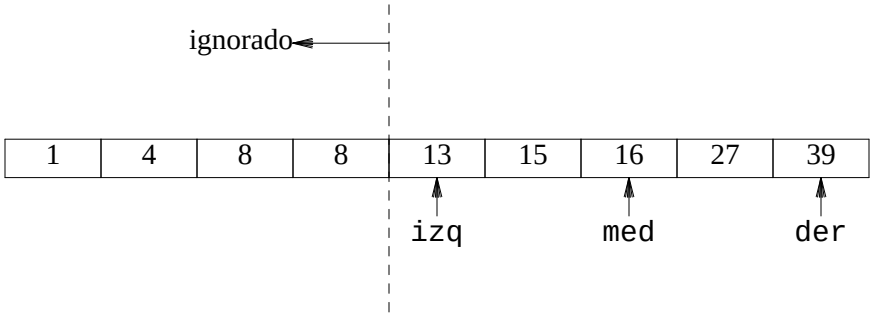


Figura 8.3: *Esquema de búsqueda binaria: partimos por la mitad en cada iteración.*

Para ver cómo funciona podemos modificar el procedimiento para que escriba los valores de *izq*, *der* y *med* en cada iteración. ¡Hazlo y fíjate en como localiza este algoritmo el valor que buscas! Prueba algún caso en que dicho valor no esté, si no no sabrás que tal caso también funciona.

8.8. Cadenas de caracteres

El tipo más popular de *array* es el conocido como **cadena de caracteres** o **string**. Un *string* es un vector que utiliza números positivos como índice (comenzando en 1) y caracteres como elementos. Este tipo de datos se utiliza para manipular texto en los programas. Por eso es tan popular. Un *string* para almacenar una palabra de 50 caracteres puede declararse como

```
type TipoPalabra is array(1..50) of Character;
```

Los *strings* son tan habituales que Ada incluye otra forma (más cómoda) de declarar *strings* y ayuda extra para inicializarlos.

La forma habitual de declarar una cadena de caracteres en Ada es utilizar la palabra reservada *string* y especificar el rango para los índices entre paréntesis a continuación. La siguiente declaración define un tipo para guardar una palabra como una cadena de 50 caracteres y declara una variable de dicho tipo:

```
type TipoPalabra is String(1..50);

palabra: TipoPalabra;
```

Podemos utilizar *palabra(1)* si queremos, para acceder al primer carácter de la palabra igual que con cualquier otro *array*.

Para expresar constantes de tipo *string* existe una sintaxis más cómoda que utilizar un agregado: escribir todos los caracteres del *array* entre comillas dobles, como hemos venido haciendo durante todo el curso. Por ejemplo, la siguiente declaración define una constante para un saludo:

```
Saludo: constant String := "hola";
```

Habrás notado que se permite omitir el rango al declarar constantes de tipo *string*, en cuyo caso el primer índice es 1 y hay tantos índices como sea preciso para almacenar el literal de tipo *string* utilizado como valor en la inicialización. Si tenemos la declaración anterior, en la memoria del ordenador tendremos un *array* de 4 caracteres como puede verse en la figura 8.4.

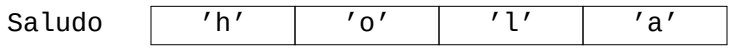


Figura 8.4: *Aspecto de una cadena de caracteres en la memoria del ordenador.*

Nótese que el tipo de datos al que pertenece el saludo anterior es distinto del tipo de datos *TipoPalabra* definido antes. El tipo al que pertenece *Saludo* es

```
String(1..4)
```

y el tipo al que pertenece *TipoPalabra* es

```
String(1..50)
```

Así, es posible asignar dos palabras entre sí pero es imposible asignarle el saludo a una palabra:

```
palabra1: TipoPalabra;
palabra2: TipoPalabra;


palabra1 := palabra2;-- bien
palabra1 := Saludo; -- *** Error ***
```

La asignación anterior entre palabras es equivalente a ejecutar el siguiente bucle:

```
for i in TipoPalabra'Range loop
      palabra1(i) := palabra2(i);
end loop;
```

Sabemos ya que se permite comparar entre sí valores de tipo *array*. Pero sólo con ‘‘=’’ y ‘‘/=’’ (de modo similar a lo que sucede con los registros). Adicionalmente, para *strings* se permiten comparaciones de orden y se utiliza el orden lexicográfico de los caracteres de los *strings* para definir el orden (similar al orden de un diccionario).

8.9. Subprogramas para arrays de cualquier tamaño

En muchas ocasiones es útil poder escribir funciones o procedimientos que sepan trabajar con *arrays* (o *strings*) de cualquier longitud. Para permitir esto es posible declarar *arrays* y *strings* sin especificar el rango de los índices cuando se utilizan como parámetros en subprogramas. A este tipo de *arrays* se les denomina **arrays sin restringir**. Por ejemplo, esta función recibe un parámetro de tipo *String* sin especificar el rango para los índices

```
function EsPalindrome(s: String) return Boolean
```

y funcionará para cualquier *string*. Eso sí, no podemos declarar variables de un tipo si sus índices no están o definidos o sugeridos por el contexto. Por ejemplo, de las dos declaraciones siguientes sólo la primera es correcta.

```
s: String := "hola";     -- Correcto
q: String;               -- Incorrecto (Tamaño?)
```

Piensa que en el caso de la función no hay mayor problema en no concretar los índices al declarar la función dado que el argumento que se suministre para el parámetro ya fijará unos límites para el *string*. En cambio, en la segunda declaración de variable mostrada arriba Ada no sabe cuánto espacio hay que reservar para *q* ni cuáles son los índices válidos; y por lo tanto no se permite esa declaración.

Aunque hemos estado utilizando *strings* como ejemplos esto puede hacerse también para cualquier tipo de *array* y no sólo para *strings*. Por ejemplo, este procedimiento es capaz de escribir un *array* de números de cualquier longitud:

```
1    type TipoNums is array(Integer range <>) of Integer;

3    procedure PutNums(nums: in TipoNums) is
4    begin
5         for i in nums'Range loop
6              Put(nums(i));
7         end loop;
8         New_Line;
9    end;
```

Y podríamos escribir...

```
Prueba1: TipoNums := (1, 2, 3, 4, 5);
 ...
PutNums(Prueba1);
```

La declaración de *TipoNums* no especifica el rango concreto para los índices y deja abierta la posibilidad que particularizar dichos índices. Eso quiere decir que no podríamos declarar una variable como la que sigue

```
Prueba2: TipoNums;  -- *** Error ***
```

dado que Ada no sabe cuánto espacio reservar ni cuáles son los índices. En cambio, la declaración que mostramos antes adquiere el tamaño de forma implícita por el tamaño del agregado utilizado para inicializarla. Podríamos también declarar:

```
Prueba3: TipoNums(1..5);
```

Esto es exactamente lo que sucede con el tipo *String*. Luego si utilizar arrays sin restringir sus límites te parece complicado, piensa que en realidad es exactamente lo mismo que hemos estado haciendo con las cadenas de caracteres.

Cuando un subprograma utiliza parámetros sin restringir puede resultar necesario declarar variables locales que adopten el mismo tipo concreto que los argumentos. Por ejemplo, una función que recibe un parámetro de tipo *String* puede querer utilizar una variable local del mismo tamaño que el argumento concreto que se le suministra. Esto puede hacerse utilizando los atributos del parámetro para ayudar en la declaración de la variable local. Piensa que, cuando se produce la llamada, el argumento sí tiene unos límites precisos y sus atributos están perfectamente definidos en ese momento. Por ejemplo, podríamos escribir algo como lo que sigue:

```
1     function Duplicar(s: String) return String is
2          doble: String(1..s'Length * 2);
3     begin
4          doble(1..s'Length) := s;
5          doble(s'Length+1..s'Length+s'Length) := s;
6          return doble;
7     end;
```

Esta función utiliza el atributo *Length* del parámetro *s* que corresponde a la longitud de *s* **una vez que se llame a la función**. Ahora mismo la función está declarada pero en realidad no existe ninguna variable *s*. Todavía. Ahora bien, si invocamos a está función de este modo:

```
Put(Duplicar("hola"));
```

Entonces, durante la llamada, el parámetro *s* es en realidad de tipo *String(1..4)* y por tanto *s'Length* es 4.

Luego la declaración de *doble* dentro de la función estará declarando una variable de tipo *String(1..8)*. El cuerpo de la función construye un string que contiene el argumento dos veces, una a continuación de otra.

8.10. ¿Es palíndrome?

Una palabra se dice que es palíndrome si se lee igual al derecho que al revés. Queremos un programa que nos diga si una palabra es palíndrome o no. Nuestro problema puede definirse como la cabecera de función:

```
function EsPalindrome(s: String) return Boolean
```

Definiéndolo así podemos resolver el problema para cualquier *string*, no sólo para los de una longitud dada.

Para ver cómo lo resolvemos podríamos aplicar directamente la definición, siguiendo con la idea de que nos inventamos cuanto podamos necesitar para poder hacer los programas *top-down*. Según la definición, una palabra palíndrome se lee igual al derecho que al revés. Si tenemos un procedimiento que invierte un *string* entonces podemos comparar el *string* original y el invertido para ver si son iguales. Luego podemos escribir...

```
1     function EsPalindrome(s: String) return Boolean is
2          sinvertido: String(s'Range);
3     begin
4          Invertir(s, sinvertido);
5          return s = sinvertido;
6     end;
```

Nos dan un *string s* y llamamos a un procedimiento (¡nuevo!) *Invertir* para que lo invierta. Aquí hemos optado por que dicho procedimiento reciba un primer argumento con el *string* que hay que invertir y devuelva en un segundo argumento el *string* ya invertido.

Al declarar nuestro *string* auxiliar llamado *sinvertido* necesitamos que su longitud sea justo la longitud de *s*. Esto lo hacemos como vimos antes: utilizamos el atributo *Range* de *s* de tal modo que, cuando se hace una llamada a *EsPalindrome*, la variable local *sinvertido* es un *string* con los mismos índices de *s*.

Pues está hecho. Si *s* es igual a *sinvertido* entonces *s* es palíndrome.

Necesitamos ahora construir un *string* que sea la versión invertida de otro. Esto es un típico problema de construcción. La idea es que recorremos el *string* original y, para cada carácter, vamos a rellenar el carácter simétrico en el nuevo *string*, tal y como muestra la figura 8.5.

En el ejemplo de la figura habría que realizar las asignaciones siguientes:

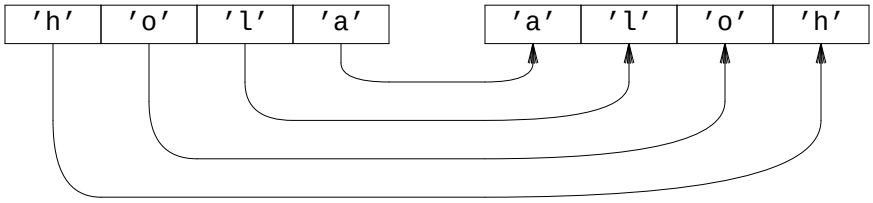


Figura 8.5: *Forma de invertir un string en otro.*

```
1     sinvertido(4) := s(1);
2     sinvertido(3) := s(2);
3     sinvertido(2) := s(3);
4     sinvertido(1) := s(4);
```

Si recorremos los índices de *s*, desde 1, basta con asignarle *s(i)* a *sinvertido(4-i+1)*.

Podríamos haber considerado primero una asignación a *sinvertido(4-i)*, pero en cuanto hubiésemos comprobado la primera y la última iteración habríamos visto que hay que ajustar el índice sumando uno. El siguiente programa incluye el procedimiento *Invertir* junto con el resto del código y algún que otro caso de prueba.

palindromes.adb

```
1     --
2     -- Ver si una palabra es palindrome
3     -- Version hecha top-down
4     --
5     with Ada.Text_IO;
6     use Ada.Text_IO;

8     procedure Palindromes is

10          package BIO is new Enumeration_IO(Boolean);
11          use BIO;

13          procedure Invertir(s: String; sinv: in out String) is
14          begin
15               for i in s'Range loop
16                    sinv(s'Last - i + 1) := s(i);
17               end loop;
18          end;

20          function EsPalindrome(s: String) return Boolean is
21               sinvertido: String(s'Range);
22          begin
23               invertir(s, sinvertido);
24               return s = sinvertido;
25          end;
```

```
27         -- Constantes de prueba
28         Prueba1: constant Boolean := EsPalindrome("1234");
29         Prueba2: constant Boolean := EsPalindrome("");
30         Prueba3: constant Boolean := EsPalindrome("abcba");
31         Prueba4: constant Boolean := EsPalindrome("abba");
32    begin
33         -- Pruebas
34         Put(Prueba1);
35         New_Line;
36         Put(Prueba2);
37         New_Line;
38         Put(Prueba3);
39         New_Line;
40         Put(Prueba4);
41         New_Line;
42    end;
—
```

Este es el resultado de ejecutarlo:

```
; ada palindromes.adb
; palindromes
FALSE
TRUE
TRUE
TRUE
```

Como podrás ver hay pruebas que consideran un *string* vacío, uno de longitud par y uno de longitud impar. La idea es intentar probar todos los casos posibles.

En realidad este programa es muy ineficiente. Recorre el *string* original construyendo otro sólo para compararlos después. Ya que hemos ganado experiencia con el problema podemos optimizarlo un poco. La idea es que podemos convertirlo en un problema similar al de comprobar si todos los números de un array son cero. Nos recorremos el *string* viendo si todos los caracteres son iguales a su carácter simétrico.

Esta es la nueva función. Al menos ahora recorremos una única vez el *string*.

```
1     function EsPalindrome(s: String) return Boolean is
2          loes: Boolean;
3          i: Integer;
4     begin
5          loes := True;
6          i := 1;
7          while i in 1..s'Last and loes loop
8               loes := s(i) = s(s'Last - i + 1);
9               i := i + 1;
10         end loop;
11         return loes;
12    end;
```

Pero, ¡Un momento! No hace falta recorrer el *string* entero. Basta comparar la primera mitad. Claro, si dicha mitad está reflejada en la segunda mitad entonces la segunda también estará reflejada en la primera mitad: no hay necesidad de volverlo a comprobar.

Esta otra función utiliza la variable *med* para marcar el punto medio del string y se limita a comparar la primera mitad.

```
1     function EsPalindrome(s: String) return Boolean is
2          med: Integer;
3          loes: Boolean;
4          i: Integer;
5     begin
6          med := (s'First + s'Last) / 2;

8          loes := True;
9          i := 1;
10         while i in 1..med and loes loop
11              loes := s(i) = s(s'Last - i + 1);
12              i := i + 1;
13         end loop;
14         return loes;
15    end;
```

Ahora tenemos una función que sólo recorre la mitad del *string*, lo que parece razonable para un programa eficiente.

8.11. Mano de cartas

Queremos un programa que nos diga el valor de una mano de cartas para el juego de las 7½. Se supone que una mano está formada por cinco cartas.

Tenemos ya programado casi todo lo que hace falta para resolver este problema. Teníamos las siguientes declaraciones que definían un tipo de datos para una carta:

```
1     -- Carta de la baraja española
2     type TipoValor is (As, Dos, Tres, Cuatro, Cinco,
3                        Seis, Siete, Sota, Caballo, Rey);

5     type TipoPalo is (Oros, Copas, Bastos, Espadas);

7     type TipoCarta is
8     record
9          valor: TipoValor;
10         palo: TipoPalo;
11    end record;
```

Si ahora queremos definir una mano podemos hacerlo sin más que definirla como un *array* de cartas. Por ejemplo:

```
constant NumCartas: Integer := 5;
  ...
type TipoMano is array(1..NumCartas) of TipoCarta;
```

La definición de nuestro problema pasa a ser:

```
function ValorMano(mano: TipoMano) return Float
```

¿Cuál es el valor de una mano? Bueno, es la suma de los valores individuales de las cartas en la mano. De nuevo nos encontramos con un problema de acumulación. Ya teníamos una función llamada *ValorCarta* que dada una carta nos daba su valor, con lo que podemos programa directamente nuestro problema siguiendo el ejemplo de la suma de números que vimos antes.

```
1     function ValorMano(mano: TipoMano) return Float is
2          valor: Float;
3     begin
4          valor := 0.0;
5          for i in 1..NumCartas loop
6               valor := valor + ValorCarta(mano(i));
7          end loop;
8          return valor;
9     end;
```

8.12. Abstraer y abstraer hasta el problema demoler

Queremos ver qué caracteres están presentes en un *string* arbitrariamente largo. ¿Cómo podemos hacer esto del modo más simple posible? Recuerda la clave de todos los problemas: podemos imaginarnos que tenemos disponible todo cuanto podamos querer, ya lo programaremos luego.

En la realidad utilizaríamos un conjunto de caracteres para resolver este problema. Podemos recorrer los caracteres del *string* e insertarlos en un conjunto (recuerda que un conjunto no tiene elementos repetidos, un elemento está o no está en el conjunto). Cuando hayamos terminado de recorrer el *string* el conjunto nos puede decir cuáles son los caracteres que tenemos. ¡Pues supongamos que tenemos conjuntos!

Por ejemplo, si lo que queremos es escribir los caracteres de nuestro *string* de prueba podríamos programar algo como esto:

```
1     procedure Caracteres is

3          ...

5          -- Constantes de prueba.
6          Prueba1: string := "una muneca pelona";

8          c: TipoCjto;
9     begin
10         NuevoCjto(c);
11         for pos in Prueba1'Range loop
12              InsertarEnCjto(c, Prueba1(pos));
13         end loop;
14         EscribirCjto(c);
15    end;
```

Nos hemos inventado de un plumazo todo lo necesario para resolver el problema. Por un lado nos declaramos alegremente una variable *c* de tipo *TipoCjto*. Esta variable va a ser un conjunto de caracteres.

Además, suponemos que llamando a *NuevoCjto* creamos un conjunto vacío en *c*. (Aunque habitualmente utilizamos funciones para crear nuevos objetos esta vez hemos optado por un procedimiento. Puede que el conjunto sea algo grande y no queremos que una función tenga que crear una copia de un conjunto nuevo sólo para asignarlo a uno que ya tenemos. Dicho de otro modo, siempre vamos a pasar nuestros conjuntos por referencia, para evitar que se copien).

El centro del programa es el bucle *for*. Este bucle recorre todo nuestro *string Prueba1* e inserta en el conjunto cada uno de sus caracteres. ¿Inserta? Desde luego: nos hemos inventado un procedimiento *InsertarEnCjto* que inserta un carácter en un conjunto de caracteres.

Y ya está. Falta imprimir el resultado del programa. Como en este caso el resultado es el valor de nuestro conjunto nos hemos inventado también una operación *EscribirCjto* que escribe un conjunto en la salida.

Problema resuelto. Salvo por... ¿Cómo hacemos el conjunto?. En este caso sabemos que sólo existen en el universo (ASCII) 256 caracteres distintos. Por lo tanto podemos utilizar un *array* para almacenar un booleano por cada carácter distinto. Si el booleano es *True* entonces el carácter está en el conjunto. Si es *False* no está.

Siempre que los elementos de un conjunto son de un tipo enumerado y sus valores posibles están confinados en un rango razonable (digamos no más de 500 o 1000 valores) entonces podemos implementar el conjunto empleando un *array* en el que los elementos serán los índices y el valor será un booleano que indica si el elemento está o no en el conjunto. Hacer esto así supone que podemos ver muy rápido si un elemento está o no está en el conjunto: basta indexar en un *array*.

Sin más dilación vamos a mostrar el programa completo, para que pueda verse lo sencillo que resulta.

caracteres.adb

```
1     with Ada.Text_IO;
2     use Ada.Text_IO;

4     procedure Caracteres is

6           subtype TipoElem is Character;

8           type TipoCjto is array(TipoElem) of Boolean;

10          procedure NuevoCjto(c: in out TipoCjto) is
11          begin
12                for e in TipoElem loop
13                      c(e) := False;
14                end loop;
15          end;

17          procedure InsertarEnCjto(c: in out TipoCjto; e: TipoElem) is
18          begin
19                c(e) := True;
20          end;

22          procedure BuscarEnCjto(c: in out TipoCjto; e: TipoElem;
23                                 esta: in out Boolean) is
24          begin
25                esta := c(e);
26          end;

28          procedure EscribirCjto(c: in out TipoCjto) is
29          begin
30                Put("[");
31                for e in TipoElem loop
32                      if c(e) then
33                            Put(e);
34                      end if;
35                end loop;
36                Put("]");
37          end;

39          -- Constantes de prueba.
40          Prueba1: string := "una muneca pelona";
```

```
42          c: TipoCjto;
43     begin
44          NuevoCjto(c);
45          for pos in Prueba1'Range loop
46               InsertarEnCjto(c, Prueba1(pos));
47          end loop;
48          EscribirCjto(c);
49     end;

51
—
```

Como se ha visto, para crear un conjunto basta con poner todos los elementos del array a *False*. ¡Esto crea el conjunto vacío!

Insertar un elemento es cuestión de poner a *True* la posición para el elemento. Y, aunque no lo necesitábamos, buscar un elemento en el conjunto es cuestión de consultar su posición en el array.

Escribir un conjunto requiere algo más de trabajo. Hay que recorrer todos los posibles elementos y ver si están o no están. Como hemos dicho que estos conjuntos (hechos con *arrays*) son de tamaño reducido (menos de 1000 elementos) no es mucho problema recorrer el *array* entero para imprimirlo.

Por si tienes curiosidad, esta es la salida del programa:

```
; ada caracteres.adb
; caracteres
[ acelmnopu]
```

Un detalle importante: no nos importa si las operaciones de nuestro flamante *TipoCjto* requieren una o mil líneas de código. Siempre vamos a definir subprogramas para ellas. De este modo podemos utilizar conjuntos de caracteres siempre que queramos sin preocuparnos de cómo están hechos. Puede que parezca poco útil si sólo vamos a utilizar el conjunto en un programa, pero no es así. Si hemos querido conjuntos de caracteres una vez es prácticamente seguro que los vamos a necesitar más veces. Además, el programa queda mucho más claro puesto que los nombres de los procedimientos y funciones dan nombre a las acciones del programa, como ya sabemos.

8.13. Conjuntos bestiales

¿Y si queremos conjuntos arbitrariamente grandes? Bueno, en realidad la pregunta es... ¿Y si queremos conjuntos cuyos elementos no estén en un rango manejable? Por ejemplo, si consideramos todos los símbolos utilizados para escribir (llamados *runas*) entonces tenemos miles de ellos. No queremos gastar un bit por cada uno sólo para almacenar un conjunto.

Si queremos no restringir el rango al que pueden pertenecer los elementos entonces tendremos que utilizar los *arrays* de otro modo. Vamos a resolver el problema del epígrafe anterior pero esta vez sin suponer que hay pocos caracteres.

Lo que podemos hacer es guardar en un *array* los elementos que sí están en el conjunto. Si utilizamos un *array* suficientemente grande entonces no habrá problema para guardar cualquier conjunto. Aunque siempre hay límites: el ordenador es una máquina finita.

El problema que tiene hacer esto así es que, aunque pongamos un límite al número de elementos que puede tener un conjunto (al tamaño del *array* que usamos para implementarlo), no todos los conjuntos van a tener justo este número de elementos.

La solución es adjuntar un entero al *array* que cuente cuántos elementos del *array* estamos utilizando en realidad. ¿No lo hemos dicho? A los enteros que utilizamos para contar cosas normalmente los denominamos **contadores**.

Este es nuestro tipo de datos para un conjunto de caracteres:

```
1     NumMaxElems: constant Integer := 500;

3     subtype TipoElem is Character;

5     type TipoElems is array(1..NumMaxElems) of Character;

7     type TipoCjto is
8     record
9          elems: TipoElems;
10         numelems: Natural;
11    end record;
```

Para crear un conjunto vacío basta con que digamos que el número de elementos es cero.

```
1     procedure NuevoCjto(c: in out TipoCjto) is
2     begin
3          c.numelems := 0;
4     end;
```

Buscar un elemento requiere recorrerse todos los elementos que tenemos en *elems*, pero sólo los *numelems* primeros (dado que son los que tenemos en realidad). Por lo demás es nuestro procedimiento de búsqueda en una colección no ordenada.

```
1     procedure BuscarEnCjto(c: in out TipoCjto; e: TipoElem;
2                            esta: out Boolean) is
3          pos: Natural;
4     begin
5          esta := False;
6          pos := 1;
7          while pos <= c.numelems and not esta loop
8               if c.elems(pos) = e then
9                    esta := True;
10              else
11                   pos := pos + 1;
12              end if;
13         end loop;
14    end;
```

Para insertar un elemento tenemos ahora una gran diferencia: hemos de mirar primero si ya está. De otro modo, si añadimos el elemento a nuestro conjunto puede que terminemos con elementos repetidos (¡Y eso no es un conjunto!). Sólo si no está lo añadimos. Añadirlo requiere incrementar el número de elementos y dejar el nuevo elemento en *elems*.

```
1     procedure InsertarEnCjto(c: in out TipoCjto; e: TipoElem) is
2          esta: Boolean;
3     begin
4          BuscarEnCjto(c, e, esta);
5          if not esta then
6               c.numelems := c.numelems + 1;
7               c.elems(c.numelems) := e;
8          end if;
9     end;
```

Nos falta escribir los elementos del conjunto. Pero eso es fácil.

```
1     procedure EscribirCjto(c: in out TipoCjto) is
2     begin
3          Put("[");
4          for pos in 1..c.numelems loop
5               Put(c.elems(pos));
6          end loop;
7          Put("]");
8     end;
```

¡Terminado! Pero no sin mencionar antes una última cosa. El programa principal que utiliza este nuevo conjunto es como sigue:

```
1     NuevoCjto(c);
2     for pos in Prueba1'Range loop
3          InsertarEnCjto(c, Prueba1(pos));
4     end loop;
5     EscribirCjto(c);
```

¡Es el mismo! Como hemos abstraído un conjunto de caracteres empleando su tipo de datos y unas operaciones que nos hemos inventado, nadie sabe cómo está hecho el conjunto. Nadie que no sea una operación del conjunto.

A partir de ahora podemos utilizar nuestro conjunto como una caja negra (sin mirar dentro). Lo mismo que hacemos con los enteros y los *arrays*. Otra cosa menos en la que pensar.

8.14. ¡Pero si no son iguales!

Resulta que al ejecutar el programa con el conjunto implementado de este segundo modo vemos que la salida es esta:

```
[una mecplo]
```

Y la salida del programa con el conjunto hecho como un *array* de booleanos era en cambio:

```
[ acelmnopu]
```

Los dos conjuntos son iguales (dado que tienen los mismos elementos). Implementa una operación *IgualCjto* que diga si dos conjuntos son iguales y pruébalo si no lo crees.

Lo que sucede es que la inserción en nuestro nuevo conjunto no es ordenada. Cada elemento se inserta a continuación de los que ya había. En el primer conjunto que implementamos los caracteres estaban ordenados dado que eran los índices del *array*.

Vamos a arreglar este problema. Podríamos ordenar los elementos del conjunto tras cada inserción. Ya sabemos cómo hacerlo. Pero parece más sencillo realizar las inserciones de tal forma que se mantenga el orden de los elementos.

La idea es que al principio el conjunto está vacío y ya está ordenado. A partir de ese momento cada inserción va a buscar dónde debe situar el elemento para que se mantengan todos ordenados. Por ejemplo, si tenemos (a,b,d,f) y queremos insertar c entonces lo que podemos hacer es desplazar d,f a la derecha en el *array* para abrir hueco para la c; y situar el nuevo elemento justo en su lugar.

Primero vamos a modificar nuestro procedimiento de búsqueda para que se detenga si sabe que el elemento no está (dado que ahora los elementos están ordenados) y para que nos diga la posición en la que se ha descubierto que el elemento está o no está. Este problema ya lo hicimos antes.

```
1     procedure BuscarEnCjto(c: in out TipoCjto; e: TipoElem;
2                            esta: out Boolean; pos: out Natural) is
3          puedeestar: Boolean;
4     begin
5          esta := False;
6          puedeestar := True;
7          pos := 1;
8          while pos <= c.numelems and not esta and puedeestar loop
9               if c.elems(pos) = e then
10                   esta := True;
11              elsif c.elems(pos) > e then
12                   puedeestar := False;
13              else
14                   pos := pos + 1;
15              end if;
16         end loop;
17    end;
```

Ahora podemos modificar *InsertarEnCjto*. La idea es que, como ya hemos dicho, si el elemento no está entonces movemos a la derecha todos los elementos que van detrás y lo insertamos justo en su sitio. El único detalle escabroso es que para mover los elementos tenemos que hacerlo empezando por el último, para evitar sobreescribir un elemento antes de haberlo movido.

```
1     procedure InsertarEnCjto(c: in out TipoCjto; e: TipoElem) is
2          pos: Natural;
3          esta: Boolean;
4     begin
5          BuscarEnCjto(c, e, esta, pos);
6          if not esta then
7               for i in reverse pos..c.numelems loop
8                    c.elems(i+1) := c.elems(i);
9               end loop;
10              c.elems(pos) := e;
11              c.numelems := c.numelems + 1;
12         end if;
13    end;
```

Problemas

1 Ver cuantas veces se repite el número 5 en un *array* de números.

2 Convertir una palabra a código morse (busca la codificación empleada por el código morse).

3 Imprimir el número que más se repite en una colección de números.

4 Ordenar una secuencia de números.

5 Ordenar las cartas de una baraja.

6 Buscar un número en una secuencia ordenada de números.

7 Imprimir un histograma para los valores almacenados en un *array*. Hazlo primero de forma horizontal y luego de forma vertical.

8 Multiplicar dos matrices de 3x3.

9 Un conjunto de enteros es una estructura de datos que tiene como operaciones crear un conjunto vacío, añadir un número al conjunto, ver si un número está en el conjunto, ver cuantos números están en el conjunto y ver el número n-ésimo del conjunto. Implementa un conjunto de enteros utilizando un array para almacenar los números del conjunto.

10 Utilizando el conjunto implementado anteriormente, haz un programa que tome una secuencia de números y los escriba en la salida eliminando números duplicados.

11 Una palabra es un anagrama de otra si tiene las mismas letras. Suponiendo que no se repite

ningún carácter en una palabra dada, haz un programa que indique si otra palabra es un anagrama o no lo es. Se sugiere utilizar de nuevo el conjunto.

12 Di si una palabra es un anagrama de otra. Sin restricciones esta vez.

13 Implementar una estructura de datos denominada Pila, en la que es posible poner un elemento sobre otros ya existentes, consular cual es el elemento de la cima de la pila y sacar un elemento de la cima de la pila. Se sugiere utilizar un array para guardar los elementos en la pila y un entero para saber cuál es la posición de la cima de la pila.

14 Utilizar la pila del problema anterior para invertir una palabra. (Basta meter todos los caracteres en una pila y luego extraerlos).

15 Calcular el valor de un número romano. Pista, dada una letra esta hay que sumarla al valor total o restarla del mismo dependiendo de si es mayor o igual a la letra que sigue o no lo es.

16 Calcular derivadas de polinomios. Para un polinomio dado calcula su derivada, imprimiéndola en la salida.

17 Implementa las operaciones de unión de conjuntos e intersección de conjuntos para las dos formas de implementar conjuntos mostradas en este capítulo.

18 Modifica la implementación del conjunto realizada como un *record* compuesto de un *array* de elementos y del número de elementos para que los elementos estén ordenados. Hazlo de dos formas: a) ordena los elementos tras cada inserción nueva; b) sitúa el nuevo elemento directamente en la posición que le corresponde en el array, tras abrirle hueco a base de desplazar el resto de elementos a la derecha.

19 Implementa un tipo de datos para manipular cadenas de caracteres de cualquier longitud (suponiendo que no habrá cadenas de más de 200 caracteres). Implementa también operaciones para asignarlas, compararlas, crearlas, añadir una cadena al final, añadirla al principio y añadirla en una posición dada.

9 — Lectura de ficheros

9.1. Ficheros

Todos los tipos de datos que hemos utilizado hasta el momento viven dentro del lenguaje. Tienen vida tan sólo mientras el programa ejecuta. Sin embargo, la mayoría de los programas necesitan leer datos almacenados fuera de ellos y generar resultados que sobrevivan a la ejecución del programa. Para esto se emplean los ficheros.

Un **fichero** es una colección de datos persistente que mantiene el sistema operativo y que tiene un nombre. También son ficheros la entrada y la salida estándar. La forma de leer de la entrada y escribir en la salida es similar a la forma de leer y escribir en cualquier fichero.

Se puede imaginar un fichero como una secuencia de records. Pero en la mayoría de los casos manipulamos **ficheros de texto** que son en realidad secuencias de caracteres. Vamos a describir únicamente como manipular ficheros de texto. Para ello utilizamos las facilidades del paquete *Text_IO*, como en realidad ya hemos estado viendo. Eso sí, a partir de ahora utilizaremos los ficheros de un modo más controlado que hasta el momento.

En Ada los ficheros están representados por el tipo de datos `File_Type`. Este tipo de datos sirve representar dentro del lenguaje algo que en realidad está fuera de el, dado que los ficheros son asunto del sistema operativo y no de ningún lenguaje de programación.

Utilizar ficheros es sencillo. Es muy similar a utilizar ficheros o archivos en el mundo real (de ahí que se llamen así, para establecer una analogía con los del mundo real). Antes de utilizar un fichero en un programa tenemos que **abrir** el fichero empleando el procedimiento *Open*. Este procedimiento prepara el fichero para leerlo o para escribirlo. Es más sencillo ver cómo utilizar *Open* viendo un ejemplo. La siguiente sentencia prepara el fichero *README.first* para leer de él, asociándolo con la variable *file* de tipo *File_Type*.

```
Open(file, In_File, "README.first");
```

El primer argumento es el fichero tal y como lo representa Ada. El segundo indica si queremos leer o escribir en el fichero. El tercer argumento es el nombre del fichero tal y como lo conoce el sistema operativo. A partir de esta sentencia podemos utilizar la variable *file* para referirnos al fichero llamado *README.first* en el ordenador. Las normas relativas a cómo se llaman los ficheros dependen del sistema operativo que se utilice y quedan fuera del ámbito de este libro.

Como nota curiosa, la entrada estándar ya está abierta desde el comienzo de la ejecución del programa sin que sea preciso invocar al procedimiento *Open*. Igual sucede con la salida. El paquete *Text_IO* incluye una variable *Standard_Input* y otra *Standard_Output*, ambas de tipo *File_Type* que corresponden a la entrada y la salida del programa.

Para guardar datos en un fichero tendremos que abrirlo para escribir en el y no para leer. Esto se hace utilizando una sentencia similar a la anterior, pero especificando la constante *Out_File* como segundo argumento de *Open* en lugar de utilizar *In_File*. Cuando lo hacemos así el fichero se vacía y se prepara para que escribamos datos en él.

Un procedimiento similar a *Open* abre el fichero para escribir en el, pero lo crea cuando no existe. Nos referimos al procedimiento *Create*. Los parámetros son los mismos que los de *Open* aunque, claro está, habitualmente se le invoca siempre especificando *Out_File*.

Una vez hemos terminado de utilizar un fichero es preciso llamar al procedimiento *Close* para **cerrar** el fichero. Por ejemplo:

```
Close(file);
```

Esto libera los recursos del ordenador que puedan utilizarse para acceder al fichero. De no llamar a dicho procedimiento es posible que los datos que hayamos escrito no estén realmente guardados

en el disco. Puesto que ni la entrada estándar ni la salida estándar las hemos abierto nosotros, tampoco hemos de cerrarlas. *Close* es sólo para ficheros que abrimos nosotros.

Del mismo modo que hemos estado utilizando procedimientos *Get* y *Put* para leer de la entrada y escribir en la salida podemos utilizar procedimientos también llamados *Get* y *Put* para leer de cualquier fichero y escribir en cualquier fichero. Igual sucede con el procedimiento *New_Line*. Para hacerlo, hay que suministrar como primer argumento (al llamarlos) una variable de tipo *File_Type* para indicar de qué fichero queremos leer o escribir. Por ejemplo, este programa abre un fichero llamado *datos.txt* y reemplaza su contenido por una única línea que contiene el texto ‘‘`hola`’’.

escribehola.adb

```
1     --
2     -- Escribe un saludo en datos.txt
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure EscribeHola is

9          fichdatos: File_Type;
10    begin
11         Open(fichdatos, Out_File, "datos.txt");
12         Put(fichdatos, "hola");
13         New_Line(fichdatos);
14         Close(fichdatos);
15    end;
—
```

Por convenio, siempre se escribe un fin de línea tras cada línea de texto. Igualmente, se supone que siempre existe un fin de línea tras cada línea de texto en cualquier fichero de texto que utilicemos como datos de entrada. Para ver cómo leer datos de un fichero que no sea la entrada podemos mirar el siguiente programa. Dicho programa lee los primeros cuatro caracteres del fichero *datos.txt* y los escribe en la salida estándar.

leehola.adb

```
1     --
2     -- Lee un saludo de datos.txt
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure LeeHola is

9          fichdatos: File_Type;
10         saludo: String(1..4);
11    begin
12         Open(fichdatos, In_File, "datos.txt");
13         Get(fichdatos, saludo);
14         Close(fichdatos);
15         Put(saludo);
16    end;
—
```

Si creamos un fichero de texto que contenga ‘‘`hola`’’ al principio y ejecutamos nuestro programa podemos ver lo que pasa:

```
; ada leehola.adb
; leehola
hola
```

Pero... ¡Cuidado!. Vamos a repetir la ejecución pero, esta vez, utilizando un fichero *datos.txt* que sólo tiene una línea de texto consistente en la palabra ‘‘no’’.

```
; leehola
raised ADA.IO_EXCEPTIONS.END_ERROR : a-textio.adb:516
```

¡El programa ha sufrido un error! Hemos intentado leer cuatro caracteres y sólo había dos que podíamos leer. Hasta el momento hemos ignorado esto en los programas que deben leer datos de la entrada, pero es hora de aprender un poco más sobre cómo controlar la lectura de la entrada en nuestros programas.

9.2. Lectura de texto

Cada fichero que tenemos abierto (incluyendo la entrada y la salida) tiene asociada una posición por la que se va leyendo (o escribiendo). Si leemos un carácter de un fichero entonces la siguiente vez que leamos leeremos lo que se encuentre tras dicho carácter. Dicho de otro modo, esta posición (conocida como **offset**) avanza conforme leemos o escribimos el fichero. A esto se le suele denominar **acceso secuencial** al fichero.

Una vez leído algo, en general, es imposible volverlo a leer. Pensemos por ejemplo en que cuando leemos de la entrada normalmente leemos lo que se ha escrito en el teclado. No sabemos qué es lo que va a escribir el usuario del programa pero, desde luego, una vez lo hemos leído ya no lo volveremos a leer. En todos los ficheros pasa algo similar: cada vez que leemos avanzamos la posición u *offset* por la que vamos leyendo, lo que hace que al leer avancemos por el contenido del fichero leyendo los datos que se encuentran a continuación de lo último que hayamos leído.

Por ejemplo, supongamos que comenzamos a ejecutar un programa y que la entrada contiene el texto:

```
Una
linea
de texto        tab
```

En tal caso la entrada del programa es la que muestra la figura 9.1. En dicha figura hemos representado con una flecha la posición por la que vamos leyendo. Esto es, la flecha corresponde al *offset* y apunta al siguiente carácter que se podrá leer del fichero. Es instructivo comparar dicha figura con el texto del fichero mostrado antes.

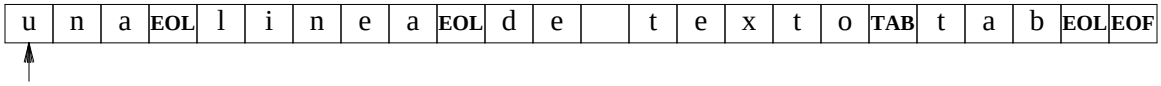


Figura 9.1: *Fichero del que lee el programa.*

Podemos ver que al final de cada línea hay una marca de fin de línea representada por **EOL** (de *end of line*) en la figura. Si del fichero anterior leemos un carácter entonces estaremos en la situación que ilustra la figura 9.2.

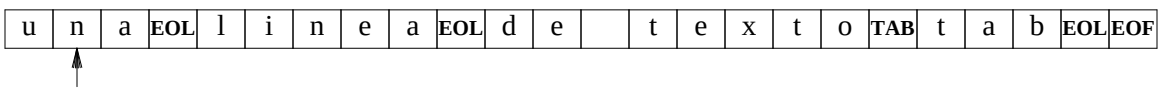


Figura 9.2: *Fichero tras leer un carácter.*

Se habrá leído el carácter ‘‘u’’ y la posición del fichero habrá avanzado un carácter. Si ahora leemos dos caracteres más entonces pasaremos a situación ilustrada en la figura 9.3.

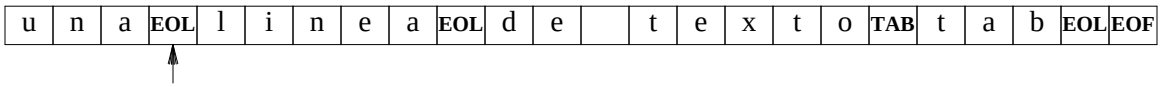
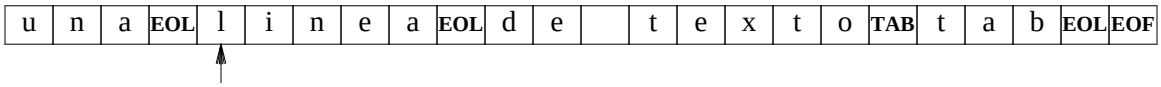


Figura 9.3: *Fichero tras haber leído dos caracteres más.*

En este momento la posición de lectura u *offset* de la entrada está apuntando al fin de línea. Si queremos seguir leyendo caracteres tenemos que evitar el fin de linea. Esta marca es en realidad uno o más caracteres que indican que se salta a otra línea en el texto. Habitualmente la escribimos pulsando la tecla *Intro*, pero no es algo que debamos leer. Leer dicha marca se considera un error, dado que no forma parte del texto que puede interesarle al programa.

Para saber si estamos a punto de leer un fin de línea podemos utilizar la función *End_Of_Line*. Dicha función devuelve *True* cuando se está mirando a la marca de fin de línea en el fichero (como sucede ahora en el ejemplo). Como de costumbre, tenemos una función llamada *End_Of_Line* sin argumentos para ver si estamos en el fin de línea en la entrada estándar y una función del mismo nombre con un parámetro de tipo *File_Type* para ver si estamos en el fin de línea del fichero suministrado como argumento.

Una vez sabemos que estamos a punto de leer un fin de línea, podemos saltarlo empleando *Skip_Line* (Sin argumentos para la entrada estándar o con un argumento de tipo *File_Type* para saltar el fin de línea en dicho fichero). Esta función salta hasta la línea siguiente y el fichero quedaría como sigue:



Podríamos seguir así indefinidamente. No obstante, no se permite leer más allá de los datos existentes en el fichero. Se dice que cuando no hay mas datos que leer de un fichero estamos en una situación de **fin de fichero**. Podemos imaginar que al final del fichero existe una marca de fin de fichero, normalmente llamada **EOF** (de *end of file*). Como dicha marca es en realidad un producto de la imaginación, es un error intentar siquiera leerla.

Para saber si estamos en una situación de fin de fichero podemos llamar a la función *End_Of_File*, que devuelve *True* cuando se está en el fin de fichero. Naturalmente, tenemos de nuevo dos funciones: una sin argumentos para la entrada estándar y otra con un argumento de tipo *File_Type* para cualquier otro fichero. Por cierto, un fichero que abrimos para escribir en el está vacío justo después de la llamada a *Open* y se encuentra en una situación de fin de fichero. Leer de el supondría un error.

En el caso de la entrada estándar puede provocarse un fin de fichero de forma artificial. En sistemas UNIX como Linux esto puede hacerse escribiendo una ‘‘d’’ mientras se pulsa la tecla *Control*. A esto se le llama normalmente *Control-d*. En el caso de sistemas Windows suele conseguirse pulsando la tecla ‘‘z’’ mientras se mantiene pulsada la tecla *Control*. A esto se le llama *Control-z*.

El siguiente programa lee el fichero *datos.txt*, evitando los fines de línea y deteniéndose en el fin de fichero, y escribe en la salida todo lo que ha leído.

copiaentrada.adb

```
1     --
2     -- Escribe el contenido de datos.txt en su salida.
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;
```

```
7     procedure CopiaEntrada is

9           fichdatos: File_Type;
10          c: Character;
11    begin
12          Open(fichdatos, In_File, "datos.txt");
13          while not End_Of_File(fichdatos) loop
14                if End_Of_Line(fichdatos) then
15                      Skip_Line(fichdatos);
16                      -- saltamos de linea
17                      New_Line;
18                else
19                      Get(fichdatos, c);
20                      -- escribimos el caracter
21                      Put(c);
22                end if;
23          end loop;
24          Close(fichdatos);
25    end;
—
```

Nótese que el programa tiene extremo cuidado de no llamar a *Get* cuando se termina el fichero. Por eso está toda la lectura encerrada en un *while* que utiliza *End_Of_File(datos)* como condición.

Otro detalle importante es que el programa no intenta siquiera leer los fines de línea presentes en el fichero. Así, si *End_Of_Line(datos)* indica que estamos mirando al fin de línea, el programa evita dicha marca de fin de línea llamando a *Skip_Line(datos)*. Sólo en otro caso se lee un carácter utilizando *Get(fichdatos,c)*.

9.3. Lectura controlada

Cuando leemos una variable de tipo *String*, *Get* consume de la entrada tantos caracteres como tiene el *string* considerado. Si, por ejemplo, el usuario escribe menos caracteres en el teclado, entonces se producirá un error en tiempo de ejecución. Así, dada la declaración

```
nombre: String(1..10);
```

la sentencia

```
Get(nombre);
```

leerá exactamente 10 caracteres, produciéndose un error si no tenemos dichos caracteres en la entrada. Esto es lo que sucedió antes cuando ejecutamos un programa para leer cuatro caracteres de un fichero que sólo tenía dos (y el fin de línea). Desde luego no queremos que eso ocurra.

Cuando llamamos a *Get* para leer enteros o números reales el procedimiento *Get* al que llamamos lee de la entrada mientras lo que lea sean blancos (espacios en blanco, tabuladores y fines de línea) y luego lee los caracteres que forman el número (deteniéndose justo en el primer carácter que no forme parte del número; ese será el siguiente carácter disponible para leer). Si llamamos a *Get* y lo que hay a la entrada no es un número, habremos modificado la posición del fichero en el intento. Y además sufriremos un error.

¿Qué podemos hacer entonces si queremos leer controladamente un fichero? Necesitamos saber qué tenemos en la entrada antes siquiera de intentar llamar a *Get*. A tal efecto existe un procedimiento llamado *Look_Ahead*. Este procedimiento devuelve dos cosas: qué carácter es el siguiente que vamos a leer y un booleano que indica si estamos a punto de leer un fin de línea. Por ejemplo, la sentencia

```
Look_Ahead(c, eol);
```

hace que *eol* sea *True* si estamos a punto de leer un fin de línea de la entrada estándar. Si ese no es el caso, la variable *c* tiene el siguiente carácter que leeremos de la entrada. Esta otra sentencia hace lo mismo pero para el fichero *fichdatos* y no para la entrada estándar:

```
Look_Ahead(fichdatos, c, eol);
```

Hay otro detalle importante. Si *Look_Ahead* nos informa de una situación de fin de línea podría ser que también estemos en una situación de fin de fichero. Esto puede ser así en el caso de la última línea del fichero.

Veamos un ejemplo de todo esto. El siguiente programa escribe en la salida estándar todo lo que pueda leer de la entrada, pero sin blancos.

sinblancos.adb

```
1     --
2     -- Escribe la entrada sin blancos en la salida.
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure SinBlancos is

9           c: Character;
10          eof: Boolean;
11          eol: Boolean;

13    begin
14          eof := False;
15          while not eof loop
16                Look_Ahead(c, eol);
17                if eol then
18                      eof := End_Of_File;
19                      if not eof then
20                            Skip_Line;
21                      end if;
22                else
23                      Get(c);
24                      if c /= ' ' and c /= ASCII.HT then
25                            Put(c);
26                      end if;
27                end if;
28          end loop;
29    end;
```

Este programa es muy importante entenderlo bien: muestra la forma de leer correctamente de la entrada estándar.

La idea es que el programa se mantendrá leyendo mientras no se haya producido un fin de fichero. Ahora bien, en lugar de utilizar directamente la función *End_Of_File* el programa utiliza *Look_Ahead*. Dependiendo de lo que averigüe dicha función el programa hará una cosa u otra.

Por un lado, puede que *Look_Ahead* diga que hay un fin de línea en la entrada. En tal caso *eol* será *True* y el programa debe ver si lo que sucede es que hay un fin de línea o si lo que sucede es que además se ha terminado el fichero (es el último fin de línea):

```
18                    eof := End_Of_File;
19                    if not eof then
20                         Skip_Line;
21                    end if;
```

Sólo se salta el fin de línea llamando a *Skip_Line* si no estamos en el fin de fichero. En otro caso, saltar el fin de línea sería un error. Recuerda que cuando estamos en el fin de fichero no deberíamos leer nada ni tampoco saltar nada. Precisamente por eso ponemos también *eof* a *True* cuando estamos ante un fin de fichero. La siguiente vez que lleguemos a la condición del *while* dejaremos de intentar procesar la entrada. ¡Se acabó!

Ahora bien, si *Look_Ahead* ha averiguado que no estamos ante un fin de línea entonces tenemos un carácter que podemos procesar.

```
23                    Get(c);
24                    if c /= ' ' and c /= ASCII.HT then
25                         Put(c);
26                    end if;
```

¡Primero hay que leerlo! Es cierto que ya sabemos lo que es `c`. Nuestro espía, *Look_Ahead*, nos lo ha dicho antes siquiera de leerlo. No obstante, para procesarlo deberíamos leerlo. Así es más fácil saber qué es lo que queda por procesar: lo que queda por leer. Además, en este caso es que no hay otra opción. Si no leemos el carácter llamando a *Get* entonces *Look_Ahead* volverá a decir lo mismo la siguiente vez que lo llamemos: crearemos un bucle infinito.

Para un programa que escribe todo lo que lee, salvo que sea un blanco, lo que tenemos que hacer es un *Put* del carácter si este no es un espacio en blanco o un tabulador (horizontal). Utilizamos la constante *ASCII.HT*, que es el carácter de tabulación horizontal.

9.4. ¿Y no hay otra forma?

Podríamos haber programado nuestro bucle principal de este modo:

sinblancos.adb

```
1     --
2     -- Escribe la entrada sin blancos en la salida.
3     -- Pero de un modo no adecuado para la
4     -- entrada estandar.
5     --
6     with Ada.Text_IO;
7     use Ada.Text_IO;

9     procedure SinBlancos is

11          c: Character;
12          eol: Boolean;
```

```
14    begin
15         while not End_Of_File loop
16              Look_Ahead(c, eol);
17              if eol then
18                   Skip_Line;
19              else
20                   Get(c);
21                   if c /= ' ' and c /= ASCII.HT then
22                        Put(c);
23                   end if;
24              end if;
25         end loop;
26    end;
—
```

Este programa también es correcto y lee cuidadosamente la entrada. No obstante, programarlo así puede dar lugar a que nos acostumbremos a escribir:

```
while not End_Of_File loop
```

A pesar de que esto en principio no tiene por qué estar mal, cuando tengamos que leer datos algo más complicados (palabras, números, etc.) es muy posible que terminemos llamando a *End_Of_File* más de una vez.

Dado que *End_Of_File* debe comprobar si hay algo más que pueda leerse o no, esta función intentará leer de la entrada (aunque no nos lo dice). Y esto es un problema: en programas donde el usuario espera que, tras un *Intro*, el programa haga algo, es muy posible que el programa esté esperando dentro de una llamada a *End_Of_File* a ver si se escribe un *Intro* o se pulsa un *Control-d*.

Prueba este programa en tu ordenador.

juegaconeof.adb

```
1     --
2     -- Juega con EOF en la entrada estandar.
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     procedure JuegaConEOF is

9     begin
10         if End_Of_File then
11              Put("EOF");
12         end if;

14         if End_Of_File then
15              Put("EOF");
16         end if;
17    end;
—
```

Si lo ejecutamos y escribimos inmediatamente un *Control-d* el programa hace esto:

```
; ada juegaconeof.adb
; juegaconeof
Control-d
EOFEOF
```

La primera llamada a *End_Of_File* ha detectado el fin de fichero y la segunda también.

Si lo ejecutamos y escribimos un *Intro* y luego provocamos un fin de fichero escribiendo *Control-d* sucede esto:

```
; juegaconeof

Control-d
E0FE0F
```

Dado que para Ada estamos en situación de fin de fichero si estamos ante el último fin de línea entonces el programa se comporta como debe.

Ahora bien, vamos a escribir *dos Intros* en lugar de uno seguidos de una indicación de fin de fichero:

```
; juegaconeof


```

¡El programa no escribe nada! Tal y como debe, claro. Pero... ¡Espera al segundo *Intro* para terminar! El primer *Intro* podría corresponder a un fin de fichero. Por lo tanto Ada se pone a leer de la entrada para ver si es el caso o no. Sólo cuando pulsamos un segundo *Intro* sabe Ada que no es el caso.

Esto quiere decir que, si queremos que el programa haga algo tras cada línea de texto que lea, no podemos llamar directamente a *End_Of_File*. Si lo hacemos, en algunas ocasiones tendremos que volver a pulsar *Intro* para que el programa haga algo (como ha pasado en la última ejecución, que requiere dos líneas para que el programa termine).

En lugar de esto es bueno acostumbrarse a utilizar *Look_Ahead*, (que sólo lee una vez, aunque nos mienta y no nos diga que ha leído). Si sabemos que estamos ante un fin de línea deberemos tener cuidado por si acaso es también el fin de fichero. Lo sea o no, es seguro que ya hemos terminado de procesar la línea de entrada con la que estábamos.

Piensa que si no es el fin de fichero, pero el usuario ha pulsado *Intro*, entonces basta una llamada a *End_Of_File* para que el programa se ponga (internamente) a leer para ver si es el fin de fichero. El usuario estará esperando a que ocurra algo pero no va a ocurrir nada hasta que escriba otra línea de entrada.

Sugerimos comprobar si estamos en una situación de fin de fichero una vez que se ha terminado de procesar la línea con la que estamos. Cuando procesamos ficheros de datos que no son la entrada estándar todo lo dicho da un poco igual (no hace falta que nadie pulse ningún *Intro*, o es el fin del fichero o no lo es). No obstante, la diferencia es abismal cuando lo que se procesa es la entrada estándar.

9.5. Separar palabras

Queremos leer controladamente la entrada para escribir a la salida una palabra por línea. Suponemos que las palabras están formadas por letras sin acentuar (esto es, ‘‘`ab33cd`’’ se consideran dos palabras: ‘‘`ab`’’ y ‘‘`cd`’’).

El problema se puede hacer leyendo carácter a carácter. El programa es muy similar al que ya hemos visto para escribir la entrada sin blancos. La diferencia es que ahora hay que saltar de línea cuando una palabra termina.

Pero no vamos a hacerlo así. Lo que queremos es utilizar nuestra idea de *top-down* y refinamiento progresivo para tener un programa capaz de leer palabras. Esto resultará muy útil. Por ejemplo, con leves cambios (que haremos luego), seremos capaces de hacer cosas tales como escribir a la salida la palabra más larga; o escribir las palabras al revés; o hacer muchas otras cosas.

Compensa con creces hacer las cosas poco a poco: no sólo se simplifica la vida, además obtenemos tipos de datos y operaciones para ellos que luego podemos reutilizar con facilidad.

Para empezar, vamos a suponer que ya tenemos un tipo de datos, *TipoPalabra*, y cuantos subprogramas podamos necesitar para manipularlo. Estando así las cosas podríamos escribir:

```
1     while not End_Of_File loop
2          SaltarBlancos;
3          if not End_Of_File then
4               LeerPalabra(palabra);
5               EscribirPalabra(palabra);
6               New_Line;
7          end if;
8     end loop;
```

Suponiendo que hemos declarado previamente:

```
palabra: TipoPalabra;
```

La idea es que para extraer todas las palabras de la entrada tenemos que, repetidamente, saltar los caracteres que separan las palabras (los llamaremos ‘‘blancos’’) y leer los que forman una palabra. Eso sí, tras saltar los blancos puede ser que estemos al final del fichero y no podamos leer ninguna otra palabra.

No está mal. Pero teniendo en cuenta lo que dijimos respecto al fin de fichero, no queremos llamar a *End_Of_File* todo el tiempo. Podemos entonces hacer que *SaltarBlancos* nos informe cuando encuentre el fin de fichero, mientras va saltando los caracteres que no forman palabras. Si lo hacemos así, el código puede quedar de este modo:

```
1     eof := False;
2     while not eof loop
3          SaltarBlancos(eof);
4          if not eof then
5               LeerPalabra(palabra);
6               EscribirPalabra(palabra);
7               New_Line;
8          end if;
9     end loop;
```

Al leer una palabra no podemos encontrarnos con el fin de fichero: *SaltarBlancos* habrá dicho que no hay *eof* y por lo tanto tendremos al menos un carácter para una palabra. Dado que suponemos que hay un fin de línea al final de cada línea, lo peor que puede pasar es que *LeerPalabra* lo encuentre y se detenga.

Para empezar a programar esto poco a poco vamos a empezar por saltarnos los caracteres que forman una palabra en lugar de leer una palabra. Nuestro programa podría quedar así:

```
1     eof := False;
2     while not eof loop
3          SaltarBlancos(eof);
4          if not eof then
5               SaltarPalabra;
6          end if;
7     end loop;
```

En este punto parece que tenemos un plan. Implementamos esto y luego complicamos un poco el programa para resolver el problema original. La forma de seguir es programar cada uno de estos subprogramas por separado. Y probarlos por separado. Basta imaginarse ahora que tenemos un conjunto de problemas (más pequeños) que resolver, cada uno diferente al resto. Los resolvemos con programas independientes y, más tarde, ya lo uniremos todo.

Para no hacer esto muy extenso nosotros vamos a implementar los subprogramas directamente, pero recuerda que las cosas se hacen poco a poco.

Empecemos por *SaltarBlancos*. Vamos a utilizar el mismo esquema que hemos utilizado antes para copiar la entrada en la salida.

```
1     procedure SaltarBlancos(eof: in out Boolean) is
2          c: Character;
3          fin: Boolean;
4          eol: Boolean;
5     begin
6          fin := False;

7          while not fin loop
8               Look_Ahead(c, eol);
9               if eol then
10                   eof := End_Of_File;
11                   fin := eof;
12                   if not fin then
13                        Skip_Line;
14                   end if;
15              elsif not EsBlanco(c) then
16                   fin := True;
17              else
18                   Get(c);
19              end if;
20         end loop;
21    end;
```

Tras llamar a *Look_Ahead* tenemos que ver no sólo si tenemos un fin de línea o no; tenemos que ver si tenemos un blanco u otra cosa. Si es un blanco hay que leerlo. Si no lo es... ¡No es nuestro! Hay que dejarlo como está: sin leer. ¿Cómo sabemos si tenemos un blanco o no? ¡Fácil! Nos inventamos una función *EsBlanco* que nos dice si un carácter es blanco o no.

Puesto que tenemos que dejar de leer cuando encontremos cualquier carácter que no sea un blanco, el bucle *while* que no utiliza *eof* como condición. Utilizamos un booleano *fin* que ponemos a *True* cuando o bien tenemos un fin de fichero o bien tenemos un carácter que no es un blanco.

Y además de todo esto tenemos que poner el parámetro *eof* a *True* si hemos encontrado un fin de fichero. Esa era la idea, ¿no?.

Un detalle aquí. Si declaramos el parámetro *eof* como *out* entonces tenemos que inicializar *eof* a *True* justo al principio del procedimiento. En lugar de eso optamos por declararlo como *in out* con lo que para nosotros dicho parámetro es, de hecho, el argumento que se ha pasado. Esto es, la responsabilidad de inicializarlo recae en el programa principal (lo que parece más natural, por otra parte).

El procedimiento *SaltarPalabra* puede ser muy similar. De hecho, es más sencillo. Sabemos que al menos tenemos un carácter para la palabra. Lo podemos leer y en función de lo que tengamos después seguimos leyendo o no.

```
1     procedure SaltarPalabra is
2          c: Character;
3          eol: Boolean;
4          fin: Boolean;
5     begin
6          loop
7               Get(c);
8               Put(c);
9               Look_Ahead(c, fin);
10              if not fin then
11                   fin := EsBlanco(c);
12              end if;
13              exit when fin;
14         end loop;
15    end;
```

Aquí, para poder ver si el programa parece hacer lo que debe o no, escribimos en la salida los caracteres que consideramos ser parte de la palabra. Al menos podemos probar el programa.

Y nos falta la función...

```
1     function EsBlanco(c: Character) return Boolean is
2     begin
3          return not (c in 'a'..'z') and not (c in 'A'..'Z');
4     end;
```

Lo tenemos hecho. Si probamos el programa tal y como está ahora veremos que hace lo mismo que el programa que eliminaba los blancos de la entrada.

¿Ahora qué? Ahora hay que leer palabras en lugar de saltarlas. Necesitamos un tipo de datos para una palabra. Una palabra es una serie de caracteres, por lo que parece que un *string* es lo más adecuado para almacenarla. La pregunta es: ¿De qué tamaño?

Podemos utilizar un *string* de un tamaño tal que nos baste para cualquier palabra que podamos encontrar y guardar, además, cuántos caracteres de dicho *string* son los que forman parte de la palabra. Empezamos por fijar el tamaño máximo de palabra que nuestro programa soportará:

```
1     MaxLongPalabra: constant Integer := 500;
```

Lo siguiente es definir un rango para el almacén de letras de la palabra y, finalmente, el *TipoPalabra*. La idea es que tenemos un almacén de letras del mismo tamaño para todas las palabras. Ahora bien, cada palabra tendrá también su longitud guardada, de manera que sabemos cuántos caracteres del almacén son válidos.

```
1     subtype TipoRangoPalabra is Integer range 1..MaxLongPalabra;

3     type TipoPalabra is
4     record
5          letras: String(TipoRangoPalabra);
6          long: Natural;
7     end record;
```

Si tenemos una palabra *pal* entonces sabemos que los caracteres de la palabra son en realidad *pal.letras(1..pal.long)*.

Por cierto, definir el tipo así no es muy distinto de definir

```
type TipoLetras is array(TipoRangoPalabra) of Character;
```

y utilizar *TipoLetras* al definir el campo *letras* del record *TipoPalabra*. Pero dado que *String* tiene algunas facilidades extra, optamos por utilizarlo.

Lo siguiente que necesitamos es poder leer una palabra. Podemos modificar nuestro procedimiento *SaltarPalabra* para que se convierta en *LeerPalabra.* Lo primero es suministrarle una palabra como argumento (por referencia). Y faltaría hacer que por cada carácter que leemos para la palabra dicho carácter se guarde en *letras*.

```
1     procedure LeerPalabra(palabra: in out TipoPalabra) is
2          c: Character;
3          eol: Boolean;
4          fin: Boolean;
5     begin
6          palabra.long := 0;    -- vacia por ahora
7          loop
8               Get(c);
9               palabra.long := palabra.long + 1;
10              palabra.letras(palabra.long) := c;
11              Look_Ahead(c, fin);
12              if not fin then
13                   fin := EsBlanco(c);
14              end if;
15              exit when fin;
16         end loop;
17    end;
```

Tras la llamada a *Get* incrementamos la longitud de la palabra; posteriormente insertamos el carácter en la posición correspondiente del array *letras*. Dicha posición es justo el valor de *palabra.long* tras el incremento, dado que el primer índice es 1.

Estamos cerca. Nos falta *EscribirPalabra*. Pero este es muy fácil.

```
1     procedure EscribirPalabra(palabra: TipoPalabra) is
2     begin
3          Put(palabra.letras(1..palabra.long));
4     end;
```

Bastaba llamar a *Put* con el trozo de *palabra.letras* que en realidad estamos usando. El programa completo quedaría como sigue.

|leerpalabras.adb|

```
1     with Ada.Text_IO;
2     use Ada.Text_IO;

4     procedure LeerPalabras is

6          MaxLongPalabra: constant Integer := 500;

8          subtype TipoRangoPalabra is Integer range 1..MaxLongPalabra;

10         type TipoPalabra is
11         record
12              letras: String(TipoRangoPalabra);
13              long: Natural;
14         end record;

16         function EsBlanco(c: Character) return Boolean is
17         begin
18              return not (c in 'a'..'z') and not (c in 'A'..'Z');
19         end;
```

```
21        procedure SaltarBlancos(eof: in out Boolean) is
22             c: Character;
23             fin: Boolean;
24             eol: Boolean;
25        begin
26             fin := False;
27             while not fin loop
28                  Look_Ahead(c, eol);
29                  if eol then
30                       eof := End_Of_File;
31                       fin := eof;
32                       if not fin then
33                            Skip_Line;
34                       end if;
35                  elsif not EsBlanco(c) then
36                       fin := True;
37                  else
38                       Get(c);
39                  end if;
40             end loop;
41        end;

43        procedure LeerPalabra(palabra: in out TipoPalabra) is
44             c: Character;
45             eol: Boolean;
46             fin: Boolean;
47        begin
48             palabra.long := 0;    -- vacia por ahora
49             loop
50                  Get(c);
51                  palabra.long := palabra.long + 1;
52                  palabra.letras(palabra.long) := c;
53                  Look_Ahead(c, fin);
54                  if not fin then
55                       fin := EsBlanco(c);
56                  end if;
57                  exit when fin;
58             end loop;
59        end;

61        procedure EscribirPalabra(palabra: TipoPalabra) is
62        begin
63             Put(palabra.letras(1..palabra.long));
64        end;

66        eof: Boolean;
67        palabra: TipoPalabra;
68   begin
69        eof := False;
70        while not eof loop
71             SaltarBlancos(eof);
72             if not eof then
73                  LeerPalabra(palabra);
74                  EscribirPalabra(palabra);
75                  New_Line;
76             end if;
77        end loop;
78   end;
```

80
—

Este programa se puede mejorar (como casi cualquier otro). Por ejemplo, como se ve al mirar *EscribirPalabra*, puede ser útil tener un valor de tipo *String* para una palabra cualquiera. Podemos programar esto como una función:

```
1     function PalabraEnAda(palabra: TipoPalabra) return String is
2     begin
3           return palabra.letras(1..palabra.long);
4     end;
```

Esto es extremadamente útil. Ahora podríamos programar *EscribirPalabra* como

```
1     procedure EscribirPalabra(palabra: TipoPalabra) is
2     begin
3           Put(PalabraEnAda(palabra));
4     end;
```

De hecho, ahora podríamos olvidarlos de programar *EscribirPalabra*. Dado que podemos convertir una palabra en un *String*, podemos utilizar cualquier subprograma que Ada tenga para ese tipo de datos, incluido *Put*.

9.6. La palabra más larga

Queremos imprimir la palabra más larga presente en la entrada del programa. De haber programado el problema anterior justo para escribir una palabra por línea, todo en un sólo procedimiento, este problema sería más complicado. Pero ahora no lo es.

Veamos. Suponiendo de nuevo que tenemos todo lo que podamos desear, podríamos programar algo como lo que sigue:

```
1     maslarga := NuevaPalabra;
2     eof := False;
3     while not eof loop
4           SaltarBlancos(eof);
5           if not eof then
6                 LeerPalabra(palabra);
7                 if LongPalabra(palabra) > LongPalabra(maslarga) then
8                       maslarga := palabra;
9                 end if;
10          end if;
11    end loop;
```

Tomamos como palabra mas larga una palabra vacía. Eso es lo que nos devolverá *NuevaPalabra*. Es suficiente, tras leer cada palabra, ver si su longitud es mayor que la que tenemos por el momento como palabra mas larga. En tal caso la palabra más larga es la que acabamos de leer.

Las dos funciones que necesitamos podemos programarlas como se ve aquí:

```
1     function LongPalabra(palabra: TipoPalabra) return Natural is
2     begin
3           return palabra.long;
4     end;
```

```
6     function NuevaPalabra return TipoPalabra is
7          pal: TipoPalabra;
8     begin
9          pal.long := 0;
10         return pal;
11    end;
```

Por lo demás no hay mucho más que tengamos que hacer, salvo escribir cuál es la palabra más larga. Para eso podríamos añadir, tras el *while* del programa principal, este código:

```
1     if LongPalabra(maslarga) = 0 then
2          Put("No hay palabras");
3     else
4          EscribirPalabra(maslarga);
5     end if;
6     New_Line;
```

Como siempre, no damos nada por sentado. Puede que no tengamos ninguna palabra en la entrada. En ese caso la palabra más larga será la palabra vacía y es mejor escribir un mensaje aclaratorio.

9.7. ¿Por qué funciones de una línea?

Esto es, ¿Por qué molestarnos en escribir funciones como *LongPalabra* que en realidad no hacen casi nada? ¿No sería mucho mejor utilizar directamente *palabra.long*?

La respuesta es simple, pero importante:

Hay que abstraer los datos.

Si tenemos un tipo de datos como *TipoPalabra* y operaciones como *NuevaPalabra*, *LeerPalabra*, *LongPalabra*, etc. entonces *nos podemos olvidar* de cómo está hecha una palabra. Esta es la clave para poder hacer programas más grandes. Si no somos capaces de **abstraer** y olvidarnos de los detalles, entonces no podremos hacer programas que pasen unos pocos cientos de líneas. Y sí, la mayoría de los programas superan con creces ese tamaño.

9.8. La palabra más repetida

Queremos averiguar cuál es la palabra que más se repite en la entrada estándar. Este problema no es igual que determinar, por ejemplo, cuál es la palabra más larga. Para saber qué palabra se repite más debemos mantener la cuenta de cuántas veces ha aparecido cada palabra en la entrada.

Lo primero que necesitamos para este problema es poder manipular palabras. Podemos ver que deberemos leer palabras de la entrada, por lo menos. De no tener implementado un tipo de datos para manipular palabras y unas operaciones básicas para ese tipo ahora sería un buen momento para hacerlo. Nosotros vamos a reutilizar dichos elementos copiándolos de los problemas anteriores.

Ahora, supuesto que tenemos nuestro *TipoPalabra*, necesitamos poder mantener una lista de las palabras que hemos visto y, para cada una de ellas, un contador que indique cuántas veces la hemos visto en la entrada. Transcribiendo esto casi directamente a Ada podemos declararnos un *array* de *records* de tal forma que en cada posición del *array* mantenemos una palabra y el número de veces que la hemos visto. A este *record* lo llamamos *TipoFrec*, puesto que sirve para ver la frecuencia de aparición de una palabra.

```
1     type TipoFrec is
2     record
3          palabra: TipoPalabra;
4          veces: Natural;
5     end record;
```

Si ponemos un límite al número máximo de palabras que podemos leer (cosa necesaria en este problema) podemos declarar entonces:

```
1     MaxNumPalabras: constant Integer := 1000;
2     subtype TipoRangoPalabras is Integer range 1..MaxNumPalabras;
3     type TipoFrecs is array(TipoRangoPalabras) of TipoFrec;
```

Pero necesitamos saber además cuántas entradas en este *array* estamos usando. Si lo pensamos, estamos de nuevo implementando una colección de elementos (como un conjunto) pero, esta vez, en lugar de caracteres estamos manteniendo elementos de tipo *TipoFreq*. Por eso vamos a llamar a nuestro tipo *TipoCjtoFreqs*.

```
1     type TipoCjtoFrecs is
2     record
3          frecs: TipoFrecs;
4          numfrecs: Natural;
5     end record;
```

Ahora que tenemos las estructuras de datos podemos pensar en qué operaciones necesitamos para implementar nuestro programa. La idea es leer todas las palabras (del mismo modo que hicimos en problemas anteriores) y, en lugar de escribirlas, insertarlas en nuestro conjunto.

```
1          NuevoCjtoFrecs(frecs);
2          eof := False;
3          while not eof loop
4               SaltarBlancos(eof);
5               if not eof then
6                    LeerPalabra(palabra);
7                    Insertar(frecs, palabra);
8               end if;
9          end loop;
```

Hemos vaciado nuestro conjunto de frecuencias (¡Que también nos hemos inventado!) con *NuevoCjtoFrecs* y utilizamos una operación *Insertar* para insertar una palabra en nuestro conjunto. La idea es que, si la palabra no está, se inserta por primera vez indicando que ha aparecido una vez. Y si ya estaba en *frecs* entonces *Insertar* se deberá ocupar de contar otra aparición, en lugar de insertarla de nuevo.

Una vez hecho esto podemos imprimir la más frecuente si nos inventamos otra operación que recupere del conjunto la palabra más frecuente:

```
1     MasFrecuente(frecs, palabra);
2     EscribirPalabra(palabra);
```

Este conjunto no tiene las mismas operaciones que el último que hicimos. En efecto, está hecho a la medida del problema. Seguramente esto haga que no lo podamos utilizar tal cual está en otros problemas.

Para insertar una palabra lo primero es ver si está. Si está incrementamos cuántas veces ha aparecido. Si no está entonces la añadimos.

```
1     procedure Insertar(c: in out TipoCjtoFrecs; palabra: TipoPalabra) is
2          pos: Natural;
3          encontrado: Boolean;
4     begin
5          Buscar(c, palabra, encontrado, pos);
6          if encontrado then
7               IncrFrec(c.frecs(pos));
8          else
9               c.numfrecs := c.numfrecs + 1;
10              c.frecs(pos) := NuevaFrec(palabra);
11         end if;
12    end;
```

Siempre es lo mismo. Nos inventamos todo lo que necesitemos. Ahora necesitamos una operación *Buscar*, otra *IncrFrec* y otra *NuevaFrec*. Nos las vamos inventando una tras otra a no ser que lo que tengamos que hacer sea tan simple que queramos escribirlo *in-situ* (si no nos importa que se manipule el tipo de datos directamente, sin utilizar operaciones).

El procedimiento para buscar es similar la los que vimos antes, salvo por un detalle: para comparar palabras no podemos comparar los *records*. Necesitamos una función *IgualPalabra* que se limite a comparar los caracteres útiles de la palabra (el *array* que utilizamos es más grande y tiene una parte que no se usa). Esta función debe comprobar antes que las palabras son de la misma longitud.

Para buscar la más frecuente es preciso tener en cuenta si el conjunto está vacío o no. Si lo está hemos optado por devolver una palabra vacía. Aunque no es muy ortodoxo, este procedimiento utiliza el valor (absurdo) 0 para *maxpos* para detectar dicho caso.

El programa completo quedaría como sigue:

masfrecuente.adb

```
1     with Ada.Text_IO;
2     use Ada.Text_IO;

4     procedure MasFrecuente is

6          MaxLongPalabra: constant Natural := 500;
7          MaxNumPalabras: constant Natural := 1000;

9          subtype TipoRangoPalabra is Natural range 1..MaxLongPalabra;
10         subtype TipoRangoPalabras is Natural range 1..MaxNumPalabras;

12         type TipoPalabra is
13         record
14              letras: String(TipoRangoPalabra);
15              long: Natural;
16         end record;

18         type TipoFrec is
19         record
20              palabra: TipoPalabra;
21              veces: Natural;
22         end record;

24         type TipoFrecs is array(TipoRangoPalabras) of TipoFrec;
```

```
26        type TipoCjtoFrecs is
27        record
28             frecs: TipoFrecs;
29             numfrecs: Natural;
30        end record;

32        function EsBlanco(c: Character) return Boolean is
33        begin
34             return not (c in 'a'..'z') and not (c in 'A'..'Z');
35        end;

37        procedure SaltarBlancos(eof: in out Boolean) is
38             c: Character;
39             fin: Boolean;
40             eol: Boolean;
41        begin
42             fin := False;
43             while not fin loop
44                  Look_Ahead(c, eol);
45                  if eol then
46                       eof := End_Of_File;
47                       fin := eof;
48                       if not fin then
49                            Skip_Line;
50                       end if;
51                  elsif not EsBlanco(c) then
52                       fin := True;
53                  else
54                       Get(c);
55                  end if;
56             end loop;
57        end;

59        function NuevaPalabra return TipoPalabra is
60             p: TipoPalabra;
61        begin
62             p.long := 0;
63             return p;
64        end;

66        function IgualPalabra(p1: TipoPalabra; p2: TipoPalabra) return Boolean is
67        begin
68             if p1.long = p2.long then
69                  return p1.letras(1..p1.long) = p2.letras(1..p1.long);
70             else
71                  return False;
72             end if;
73        end;
```

```
75        procedure LeerPalabra(p: out TipoPalabra) is
76             c: Character;
77             eol: Boolean;
78             fin: Boolean;
79        begin
80             p.long := 0;    -- vacia por ahora
81             loop
82                  Get(c);
83                  p.long := p.long + 1;
84                  p.letras(p.long) := c;
85                  Look_Ahead(c, fin);
86                  if not fin then
87                       fin := EsBlanco(c);
88                  end if;
89                  exit when fin;
90             end loop;
91        end;

93        procedure EscribirPalabra(p: TipoPalabra) is
94        begin
95             Put(p.letras(1..p.long));
96        end;

98        function NuevaFrec(p: TipoPalabra)  return TipoFrec is
99             frec: TipoFrec;
100       begin
101            frec.palabra := p;
102            frec.veces := 1;
103            return frec;
104       end;

106       procedure IncrFrec(frec: in out TipoFrec) is
107       begin
108            frec.veces := frec.veces + 1;
109       end;

111       procedure NuevoCjtoFrecs(c: out TipoCjtoFrecs) is
112       begin
113            c.numfrecs := 0;
114       end;

116       procedure Buscar(c: in out TipoCjtoFrecs; p: TipoPalabra;
117                        encontrado: out Boolean; pos: out Natural) is
118       begin
119            encontrado := False;
120            pos := 1;
121            while pos <= c.numfrecs and not encontrado loop
122                 if IgualPalabra(c.frecs(pos).palabra, p) then
123                      encontrado := True;
124                 else
125                      pos := pos + 1;
126                 end if;
127            end loop;
128       end;
```

```
130         procedure Insertar(c: in out TipoCjtoFrecs; palabra: TipoPalabra) is
131              pos: Natural;
132              encontrado: Boolean;
133         begin
134              Buscar(c, palabra, encontrado, pos);
135              if encontrado then
136                   IncrFrec(c.frecs(pos));
137              else
138                   c.numfrecs := c.numfrecs + 1;
139                   c.frecs(pos) := NuevaFrec(palabra);
140              end if;
141         end;

143         procedure MasFrecuente(c: in out TipoCjtoFrecs; palabra: out TipoPalabra) is
144              maxpos: Natural;
145              maxveces: Natural;
146         begin
147              maxveces := 0;
148              maxpos := 0;

150              palabra := NuevaPalabra;
151              for i in 1..c.numfrecs loop
152                   if c.frecs(i).veces > maxveces then
153                        maxpos := i;
154                        maxveces := c.frecs(i).veces;
155                   end if;
156              end loop;
157              if maxpos /= 0 then
158                   palabra := c.frecs(maxpos).palabra;
159              end if;
160         end;

162         eof: Boolean;
163         palabra: TipoPalabra;
164         frecs: TipoCjtoFrecs;
165    begin
166         NuevoCjtoFrecs(frecs);
167         eof := False;
168         while not eof loop
169              SaltarBlancos(eof);
170              if not eof then
171                   LeerPalabra(palabra);
172                   Insertar(frecs, palabra);
173              end if;
174         end loop;
175         MasFrecuente(frecs, palabra);
176         EscribirPalabra(palabra);
177    end;

179
```

—

Problemas

1 Implementar un procedimiento que lea una línea de la entrada, devolviendo tanto los caracteres leídos como el número de caracteres leídos. Se supone que dicha línea existe en la entrada.

2 Leer palabras de la entrada estándar e imprimir la más larga. Suponiendo que en la entrada

hay una palabra por línea.

3 Convertir la entrada estándar a código morse (busca la codificación empleada por el código morse).

4 Imprimir el número que más se repite en la entrada estándar.

5 Imprimir la palabra que más se repite en la entrada estándar. Suponiendo que en la entrada hay una palabra por línea.

6 Imprimir las palabras de la entrada estándar al revés. Suponiendo que en la entrada hay una palabra por línea.

7 Imprimir las palabras de la entrada estándar de la mas corta a la más larga. Suponiendo que en la entrada hay una palabra por línea.

8 Imprimir un histograma para la frecuencia de aparición de los valores enteros leídos de la entrada estándar. Primero de forma horizontal y luego de forma vertical.

9 Leer un laberinto de la entrada estándar expresado como un array de arrays de caracteres, donde un espacio en blanco significa hueco libre y un asterisco significa muro.

10 Almacenar datos de personas leyéndolos de la entrada estándar. Para cada persona hay que almacenar nombre y DNI. En la entrada hay una persona por línea.

11 Completar el problema anterior de tal forma que sea posible buscar personas dado su DNI.

12 Convertir la entrada estándar en mayúscula utilizando un array para hacer la traducción de letra minúscula a letra mayúscula. Los caracteres que no sean letras deben quedar como estaban.

13 Leer un vector de palabras de la entrada estándar. Se supone que las palabras son series de letras minúsculas o mayúsculas y que las palabras estan separadas por cualquier otro carácter. No pueden hacerse otras suposiciones sobre la entrada y no puede utilizarse *Get_Line*.

14 Realizar el programa anterior para leer los datos del fichero *datos.txt*.

15 Justificar el fichero *datos.txt* a izquierda y derecha, empleando líneas de 80 caracteres y márgenes de 5 caracteres a izquierda y derecha. Imprimir el resultado de la justificación en la salida estándar. Se supone que las palabras son series de caracteres separados por blancos o signos de puntación. No pueden hacerse otras suposiciones sobre la entrada y no puede utilizarse *Get_Line*.

16 Implementar una calculadora que además de expresiones aritméticas simples pueda evaluar funciones de un sólo argumento tales como *abs*, *sin*, y *cos*. La calculadora debe leer expresiones del fichero *datos.txt* hasta el fin de fichero y distinguir correctamente las operaciones infijas de las funciones. Debe funcionar correctamente independientemente de cómo se escriban las expresiones (respecto a caracteres en blanco). Aunque puede suponerse que todas las expresiones son correctas. Todos los números leídos son reales. No pueden hacerse otras suposiciones sobre la entrada y no puede utilizarse *Get_Line*.

17 Añadir dos memorias a la calculadora anterior. Si a la entrada se ve

```
mem 1
```

el último valor calculado se almacena en la memoria 1. Igualmente para la memoria 2. Si a la entrada se ve ‘‘`R1`’’ se utiliza el valor de la memoria 1 en lugar de ‘‘`R1`’’. Igualmente para la memoria 2.

18 Implementar un evaluador de notación polaca inversa utilizando la pila realizada en un problema anterior. Si se lee un numero de la entrada se inserta en la pila. Si se lee una operación de la entrada (un signo aritmético) se sacan dos números de la pila y se efectúa la operación, insertando el resultado de nuevo en la pila. Se imprime cada valor que se inserta en la pila. Por ejemplo, si se utiliza ‘‘`2 1 + 3 *`’’ como entrada el resultado ha de ser 9. Los números son enteros siempre. No pueden hacerse suposiciones sobre la entrada y no

puede utilizarse *Get_Line*. La entrada del programa ha de tomarse del fichero *datos.txt*.

19 Calcular derivadas de polinomios. Lee una serie de polinomios de la entrada y calcula sus derivadas, imprimiéndolas en la salida.

20 Implementar un programa sencillo que lea cartones de bingo. Se supone que tenemos cartones que consisten en dos líneas de cinco números. Los números van del 1 al 50 y no se pueden repetir en el mismo cartón. Podemos tener uno o mas cartones. El programa debe leer las líneas correspondientes a los cartones de la entrada (podemos tener un máximo de 5 cartones) hasta que encuentre la palabra ‘‘`fin`’’. (Puede haber otras palabras que hay que indicar que son órdenes no reconocidas). La entrada puede incluir los números en cualquier orden y utilizando cualquier número de líneas y espacios en blanco pero podemos suponer que están todos los números para cada cartón y no falta ni sobra ninguno. No pueden hacerse otras suposiciones sobre la entrada y no puede utilizarse *Get_Line*. La entrada del programa ha de tomarse del fichero *datos.txt*.

21 Continuando el ejercicio anterior, implementar un programa para jugar al bingo. Una vez leídos los cartones del fichero *datos.txt*, el programa debe leer números de la entrada estándar e indicar cuando se produce *línea* (todos los números de una línea han salido) imprimiendo el número del cartón (el primero leído, el segundo, etc.) e imprimiendo el número de la línea. También hay que indicar cuando se produce *bingo* (todos los números de un cartón han salido) y terminar el juego en tal caso. Sólo puede leerse carácter a carácter y naturalmente hay que contemplar el caso en que el usuario, por accidente, escribe el mismo número más de una vez (dado que un número sólo puede salir una vez). Tras leer cada cartón hay que escribir el cartón correspondiente. Tras cada jugada hay que escribir todos los cartones.

22 Modifica los programas realizados anteriormente para que todos ellos lean correctamente de la entrada. Esto es, no deben hacer suposiciones respecto a cuanto espacio en blanco hay en los textos de entrada ni respecto a dónde se encuentra este (salvo, naturalmente, por considerar que debe haber espacio en blanco para separar palabras que no puedan separarse de otro modo).

23 Modifica la calculadora del ejercicio 16, una vez hecho el ejercicio 15 del capítulo anterior, para que sea capaz de utilizar también números romanos como argumentos de las expresiones y funciones. Eso sí, los números romanos se supone que siempre se escriben en mayúsculas.

10 — Haciendo programas

10.1. Calculadora

Queremos construir una calculadora de las llamadas de línea de comandos. Debe ser un programa que lea de la entrada estándar una serie de expresiones aritméticas e imprima el valor de las mismas en la salida. La intención es utilizarla como calculadora de sobremesa.

Además de las expresiones aritméticas habituales (suma, resta, multiplicación y división) también queremos que la calculadora sea capaz de aceptar funciones matemáticas elementales. Por ejemplo: senos, cosenos, raíces cuadradas, etc.

Como la calculadora la queremos utilizar en viajes de carácter turístico queremos que sea capaz de entender no sólo la numeración arábica básica, sino también números romanos.

Como facilidad extra deseamos que la calculadora tenga varias memorias, para que podamos utilizarlas para recordar números durante el cálculo de expresiones.

10.2. ¿Cuál es el problema?

A estas alturas conocemos todo lo necesario para implementar este programa. Lo primero es ver si entendemos el problema. Todo lo que no nos dice el enunciado queda a nuestra elección (aunque habrá que hacer elecciones razonables). Vamos a leer expresiones de la entrada estándar y, para cada una de ellas, calcular el valor resultante e imprimirlo en la salida estándar. Por ejemplo, podríamos leer algo como:

```
3 + 2
```

Podríamos considerar expresiones más complicadas, del estilo a

```
3 + 2 * 5
```

pero esto queda fuera del ámbito de este curso y lo dejamos propuesto como una futura mejora. Tan importante como elegir qué implementar es elegir qué no implementar. En cualquier caso parece que debemos entender cosas como

```
seno 3
```

También tenemos que poder aceptar números romanos. Como en...

```
coseno MCM
3.5 / XIII
```

En cuanto a las memorias de la calculadora, podemos ver cómo funcionan en las calculadoras que conocemos. Normalmente hay una función que almacena el último valor calculado y luego tenemos otra que recupera ese valor. Nosotros vamos a utilizar

```
M1
```

para almacenar el último valor en la memoria 1 y

```
M2
```

para almacenarlo en la memoria 2. Para recuperar el valor de la memoria 1 podríamos utilizar `R1`, como en

```
3 / R1
```

Por lo que parece esto es todo.

10.3. ¿Cuál es el plan?

Tenemos que dividir el problema en subproblemas hasta que todos ellos sean triviales. Ese es el plan. Como vemos, nuestra calculadora debe seguir básicamente este algoritmo:

```
while not End_Of_File loop
      LeerExpr(expr);
      valor := EvaluarExpr(expr);
      Put(valor);
end loop;
```

Esto es pseudocódigo, pero mejor hacerlo cuanto más detallado mejor.

La dificultad del problema radica en que hay muchos tipos de expresiones distintas y tenemos funciones, memorias, etc. Por lo tanto, empezaremos por implementar la versión mas simple de calculadora que se nos ocurra (dentro de la especificación del problema).

Aunque hagamos esta simplificación intentaremos que el código y los datos que utilicemos sean generales, de tal forma que luego podamos irlos modificando y uniendo hasta tener resuelta la totalidad del problema original.

Empezaremos por evaluar expresiones aritméticas sencillas; consistentes siempre en un operando (un número), un operador y un segundo operando.

Posteriormente podríamos añadir funciones a nuestra calculadora, después números romanos y, para terminar, memorias. Como parece que este plan de acción es viable es mejor dejar de pensar en cómo hacer las cosas y ponerse a hacerlas.

10.4. Expresiones aritméticas

Vamos a transformar el pseudocódigo de arriba en código. Lo primero que necesitamos es un tipo de datos para los objetos que ha de manipular el programa: expresiones aritméticas.

Una expresión aritmética es una operación y dos operandos (en nuestra versión inicial simplificada del problema). Los operandos serán números reales y los operadores podemos considerar que serán: suma, resta, multiplicación y división. Si luego queremos añadir más, basta con hacerlo.

Las operaciones van a ser elementos del tipo *TipoOp*:

```
type TipoOp  is (Suma, Resta, Multiplicacion, Division);
```

Los argumentos (dado que son varios) parece razonable que sean un *array* de números reales. ¡Cuidado con la tentación de definirlos como dos números separados! Si lo hacemos así entonces para el programa serán dos cosas separadas. Sin embargo, los estamos llamando ‘‘los argumentos’’. Bueno, pues necesitamos un *TipoArgs*:

```
MaxNumArgs: constant Natural := 2;
type TipoArgs is array(1..MaxNumArgs) of Float;
```

Un programa nunca debe contener números ‘‘mágicos’’. Por ejemplo, si nuestro *array* utiliza el rango *1..2* podemos preguntarnos inmediatamente ¿Por qué 2 y no 5? Siempre tenemos que definir constantes, de tal modo que cada uno de los parámetros o constantes que definen el problema pueda cambiarse, sin más que cambiar el valor de una constante en el programa.

¡Hora de compilar y ejecutar nuestro primer prototipo del programa!

calc.adb

```
1     --
2     -- Calculadora de linea de comandos
3     -- prototipo 1
4     --
```

```
6     with Ada.Text_IO;
7     use Ada.Text_IO;

9     procedure Calc is

11          MaxNumArgs: constant Natural := 2;

13          type TipoOp  is (Suma, Resta, Multiplicacion, Division);

15          type TipoArgs is array(1..MaxNumArgs) of Float;

17          -- expresion aritmetica de tipo "3 + 2"
18          type TipoExpr is
19          record
20               op: TipoOp;
21               args: TipoArgs;
22          end record;

24    begin
25          null;
26    end;
—
```

```
; ada calc.adb
; calc
```

Podemos ahora pensar en leer una expresión y evaluarla. Pero antes incluso de eso deberíamos al menos poder crear una expresión e imprimirla (aunque sólo sea para depurar errores que podamos cometer).

Para crear una nueva expresión necesitamos una operación y dos argumentos y devolvemos una expresión. Parece fácil.

```
1     function NuevaExpr(op: TipoOp; arg1: Float; arg2: Float) return TipoExpr is
2          expr: TipoExpr;
3     begin
4          expr.op := op;
5          expr.args(1) := arg1;
6          expr.args(2) := arg2;
7          return expr;
8     end;
```

Podríamos haber suministrado un *array* de argumentos, pero queremos que la función nos deje crear una nueva expresión de una forma fácil y rápida. Si más adelante la función no nos sirve o hemos de cambiarla, ya nos ocuparemos cuando llegue el momento.

En cualquier caso, es hora de compilar y ejecutar nuestro programa.

```
; ada calc.adb
```

Para imprimir una expresión basta con imprimir los argumentos y la operación en la forma habitual. En este caso resultará útil suponer que tenemos disponible una función `EscribirOp`, cosa que haremos.

```
1     procedure EscribirExpr(expr: TipoExpr) is
2     begin
3          Put(expr.args(1));
4          EscribirOp(expr.op);
5          Put(expr.args(2));
6     end;
```

Y ahora necesitamos...

```
1     procedure EscribirOp(op: TipoOp) is
2     begin
3          case op is
4          when Suma =>
5               Put("+");
6          when Resta =>
7               Put("-");
8          when Multiplicacion =>
9               Put("*");
10         when Division =>
11              Put("/");
12         end case;
13    end;
```

Se podría haber utilizado un *array* para obtener el *string* correspondiente al nombre de una operación, en lugar de utilizar una función. Hazlo tú como prueba y compara el resultado.

Podemos crear ahora algunas expresiones e imprimirlas, para ver qué tal vamos. Empezamos por definir las constantes y poner alguna sentencia en el cuerpo del programa:

```
1          -- constantes de pruebas
2          Prueba1: constant TipoExpr := NuevaExpr(Suma, 3.0, 2.2);
3          Prueba2: constant TipoExpr := NuevaExpr(Division, 3.2, 0.0);
4     begin
5          EscribirExpr(Prueba1);
6          New_Line;
7          EscribirExpr(Prueba2);
8          New_Line;
9     end;
```

Y ahora lo probamos:

```
; ada calc.adb
; calc
 3.00000E+00+ 2.20000E+00
 3.20000E+00/ 0.00000E+00
```

La suma y la resta salen junto al primer operando y necesitamos algún espacio en blanco antes de escribir la operación. Hay que modificar *EscribirExpr*. No cambiamos *EscribirOp* para incluir un espacio en blanco. En su lugar hacemos esto otro:

```
1     procedure EscribirExpr(expr: TipoExpr) is
2     begin
3          Put(expr.args(1));
4          Put(" ");
5          EscribirOp(expr.op);
6          Put(expr.args(2));
7     end;
```

Piensa que *EscribirOp* debe escribir una operación y nada más.

Una cosa más, dado que ya hemos probado estos subprogramas (aunque no mucho) es hora de borrar del programa las constantes de pruebas y volver a dejar en el programa principal:

```
1     begin
2          null;
3     end;
```

10.5. Evaluación de expresiones

Tenemos muy a mano evaluar las expresiones que podemos construir (e imprimir). Todo depende de la operación y de los argumentos. Así pues, podemos escribir una función que calcule el valor de una expresión aritmética.

```
1     function EvalExpr(expr: TipoExpr) return Float is
2          result: Float;
3     begin
4          case expr.op is
5          when Suma =>
6               result := expr.args(1) + expr.args(2);
7          when Resta =>
8               result := expr.args(1) - expr.args(2);
9          when Multiplicacion =>
10              result := expr.args(1) * expr.args(2);
11         when Division =>
12              result := expr.args(1) / expr.args(2);
13         end case;
14         return result;
15    end;
```

Hora de probarlo. Escribimos de nuevo alguna prueba...

```
1          -- constantes de pruebas
2          Prueba1: constant TipoExpr := NuevaExpr(Suma, 3.0, 2.2);
3          Prueba2: constant TipoExpr := NuevaExpr(Division, 3.2, 0.0);
4          Valor1: constant Float := EvalExpr(Prueba1);
5          Valor2: constant Float := EvalExpr(Prueba2);
6     begin
7          Put(Valor1);
8          New_Line;
9          Put(Valor2);
10         New_Line;
11    end;
```

Y la ejecutamos:

```
; ada calc.adb
; calc
 5.20000E+00
+Inf********
```

El segundo valor de prueba resulta ser un error: hemos dividido por cero. En muchos lenguajes intentar dividir por cero detiene la ejecución del programa. Vamos a modificar *EvalExpr* para que evite dividir por cero. Si escribimos algo como

```
if expr.args(2) = 0 then
      ????
else
      result := expr.args(1) / expr.args(2);
end if;
```

entonces tenemos el problema de qué valor devolver cuando el segundo argumento es cero. En lugar de esto vamos a hacer que el segundo argumento sea muy pequeño, pero no cero. Esta es la nueva función *EvalExpr*.

```
1     function EvalExpr(expr: TipoExpr) return Float is
2          result: Float;
3          arg2: Float;
4     begin
5          case expr.op is
6          when Suma =>
7               result := expr.args(1) + expr.args(2);
8          when Resta =>
9               result := expr.args(1) - expr.args(2);
10         when Multiplicacion =>
11              result := expr.args(1) * expr.args(2);
12         when Division =>
13              if expr.args(2) = 0.0 then
14                   arg2 := 1.0e-30;
15              else
16                   arg2 := expr.args(2);
17              end if;
18              result := expr.args(1) / arg2;
19         end case;
20         return result;
21    end;
```

Si la probamos ahora parece que hace algo razonable (para este tipo de calculadora)

```
; ada calc.adb
; calc
 5.20000E+00
 3.20000E+30
```

En adelante no lo repetiremos más por razones de brevedad. No obstante, recuerda que cada cosa que se incluye en el programa se prueba inmediatamente. Si no se puede probar en el programa por tener algo a medio hacer entonces se hace otro programa sólo para probarlo. Si para poderlo probar necesitamos otras operaciones más básicas entonces las programamos con un cuerpo que no haga nada (o devuelva siempre cero, o lo que haga falta) para que podamos al menos compilar y ejecutar lo que estamos programando.

10.6. Lectura de expresiones

Lo siguiente puede ser leer expresiones de la entrada. Esto tenemos que hacerlo de forma controlada. Por el momento la entrada será algo como

```
3 + 2
```

pero sabemos que dentro de poco serán cosas como

```
seno 3
```

y no podemos simplemente utilizar *Get* para leer un número. Si no hay un número tendremos un problema.

Por fortuna, las expresiones que tenemos implementadas siempre empiezan por un carácter numérico. El resto de expresiones que tendrá que manipular la calculadora empezarán por letras. Por el momento podemos leer siempre un número y luego una operación seguida por otro número. En el futuro habrá que ver qué hay en la entrada y leer una cosa u otra en función de lo que encontremos.

En cualquiera de los casos el usuario puede equivocarse al escribir y puede que encontremos expresiones erróneas. ¿Qué vamos a hacer si eso ocurre?

Empezaremos por escribir nuestro programa principal. Podría ser algo como esto.

```
1          eof: Boolean;
2          expr: TipoExpr;
3          valor: Float;
4     begin
5          eof := False;
6          while not eof loop
7               LeerExpr(expr, eof);
8               if not eof then
9                    valor := EvalExpr(expr);
10                   Put(valor);
11                   New_Line;
12              end if;
13         end loop;
14    end;
```

Recuerda que no queremos utilizar directamente *End_Of_File*, debido a que en programas interactivos como este eso puede dar lugar a que el programa se quede esperando (dentro de *End_Of_File*) a que el usuario pulse más veces la tecla *Intro* antes de hacer nada con la línea que ya ha leído. Así pues, parece mejor que *LeerExpr* se ocupe de esos asuntos y le diga al programa que lo utiliza si se ha encontrado el fin de fichero o no. Si no se lo ha encontrado suponemos que tenemos una expresión que evaluar: la evaluamos, imprimimos el resultado y volvemos a empezar.

Para leer una expresión tenemos que hacer algo muy parecido a lo que hicimos en un capítulo anterior: saltar blancos y leer palabras. Ahora saltamos blancos y leemos una expresión. El procedimiento *SaltarBlancos* ya lo implementamos en el capítulo anterior y lo omitimos aquí. La función *EsBlanco* utilizada por dicho procedimiento también tenemos que tomarla prestada. Pero cuidado: ahora un carácter lo consideramos blanco cuando es un blanco (espacio o tabulador). Nuestra función para leer una expresión podría entonces ser como sigue.

```
1     procedure LeerExpr(expr: out TipoExpr; eof: in out Boolean) is
2     begin
3          SaltarBlancos(eof);
4          if not eof then
5               LeerNumero(expr.args(1));
6          end if;
7          if not eof then
8               SaltarBlancos(eof);
9          end if;
10         if not eof then
11              LeerOp(expr.op);
12         end if;
13         if not eof then
14              SaltarBlancos(eof);
15         end if;
16         if not eof then
17              LeerNumero(expr.args(2));
18         end if;
19    end;
```

Tenemos que saltarnos el espacio en blanco que pueda haber entre números y operaciones. Y, en cualquier caso, tenemos que dejar de leer si nos encontramos el fin de fichero. Eso sí, si tras saltar los blancos no nos hemos encontrado con el fin de fichero entonces seguro que tenemos al menos un carácter no blanco para la siguiente palabra que tengamos a la entrada (sea un número o una operación). Además, la lectura de un número o de una operación se detendrá en el primer carácter que no pueda formar parte del número o de la operación. Por eso no es preciso que ni *LeerNumero* ni *LeerOp* devuelvan ninguna información respecto al fin de fichero.

Otro detalle es que hemos optado por rodear cada operación por un *if* separado (en lugar de anidar los *ifs*). Haciéndolo de este modo el código se lee mejor; al precio de comprobar si *eof* es *True* múltiples veces. Compensa pagar el precio. De no haberlo hecho así, *LeerExpr* habría quedado como sigue:

```
1     procedure LeerExpr(expr: in out TipoExpr; eof: in out Boolean) is
2     begin
3          SaltarBlancos(eof);
4          if not eof then
5               LeerNumero(expr.args(1));
6               SaltarBlancos(eof);
7               if not eof then
8                    LeerOp(expr.op);
9                    SaltarBlancos(eof);
10                   if not eof then
11                        LeerNumero(expr.args(2));
12                   end if;
13              end if;
14         end if;
15    end;
```

Compara este código con el anterior. Cuesta mucho trabajo ver lo que hace y en cambio, comprobando *eof* algunas veces más, el código anterior deja clara la secuencia de acciones que está ejecutando.

Como esta es una calculadora sencilla vamos a suponer que los números que escribe el usuario son de la forma:

- Opcionalmente un signo ‘‘+’’ o ‘‘-’’.
- Una serie de dígitos.
- Y, opcionalmente, un ‘‘.’’ y más dígitos.

La idea es partir con el valor real a cero. Conforme leamos dígitos podemos multiplicar el valor acumulado por 10 y sumar el dígito leído. Eso se ocupa de leer la parte entera. Si hay parte decimal hacemos algo parecido; en este caso tenemos que dividir cada dígito decimal por sucesivas fracciones de 10 y sumarlo al valor total.

Para leer la parte entera sin signo podríamos programar algo como esto:

```
1     procedure LeerNumero(num: out Float) is
2          c: Character;
3          digito: Integer;
4          fin: Boolean;
5     begin
6          loop
7               Get(c);
8               digito := Character’Pos(c) - Character’Pos(’0’);
9               num := num * 10.0 + Float(digito);
10              Look_Ahead(c, fin);
11              if not fin then
12                   fin := not EsDigito(c);
13              end if;
14              exit when fin;
15         end loop;
16    end;
```

Dejamos de leer justo cuando dejamos de tener un dígito. Y suponemos que tenemos un dígito a la entrada (quien nos llame tendrá que saltar él los blancos). Ahora bien, si tenemos parte decimal vamos a tener que hacer casi lo mismo (salvo por la forma de acumular el valor). Así que podemos hacer esto otro:

```
1     procedure LeerNumero(num: out Float) is
2          c: Character;
3          digito: Integer;
4          fin: Boolean;
5          parteentera: Boolean;
6          fraccion: Float;
7     begin
8          num := 0.0;
9          parteentera := True;
10         fraccion := 1.0;

12         loop
13              Get(c);
14              digito := Character'Pos(c) - Character'Pos('0');
15              if parteentera then
16                   num := num * 10.0 + Float(digito);
17              else
18                   fraccion := fraccion / 10.0;
19                   num := num + Float(digito) * fraccion;
20              end if;

22              Look_Ahead(c, fin);
23              if not fin then
24                   if c = '.' then
25                        parteentera := False;
26                        Get(c);
27                   end if;
28              end if;

30              Look_Ahead(c, fin);
31              if not fin then
32                   fin := not EsDigito(c);
33              end if;
34              exit when fin;
35         end loop;
36    end;
```

Pero nos falta tener en cuenta el signo. Lo mejor es utilizar un *case* y hacer una cosa u otra en función del carácter que tenemos entre manos. Si el carácter nos gusta lo consumimos. Si no hemos terminado.

```
1     procedure LeerNumero(num: out Float) is
2          c: Character;
3          digito: Integer;
4          fin: Boolean;
5          parteentera: Boolean;
6          fraccion: Float;
7          signo: Float;
8     begin
9          num := 0.0;
10         parteentera := True;
11         fraccion := 1.0;
12         signo := 1.0;
```

```
14          Look_Ahead(c, fin);
15          while not fin loop
16               case c is
17               when '-' =>
18                    signo := -1.0;
19                    Get(c);
20               when '+' =>
21                    Get(c);
22               when '.' =>
23                    parteentera := false;
24                    Get(c);

26               when '0'..'9' =>
27                    Get(c);
28                    digito := Character'Pos(c) - Character'Pos('0');
29                    if parteentera then
30                         num := num * 10.0 + Float(digito);
31                    else
32                         fraccion := fraccion / 10.0;
33                         num := num + Float(digito) * fraccion;
34                    end if;

36               when others =>
37                    fin := True;
38               end case;

40               if not fin then
41                    Look_Ahead(c, fin);
42               end if;
43          end loop;
44          num := signo * num;
45     end;
```

Estamos cerca. Nos falta leer una operación. Esto es simplemente mirar si el carácter es una de las operaciones que conocemos. ¿Qué haremos si no es ninguna de ellas? Podemos modificar nuestro enumerado de operaciones e inventarnos una operación llamada *Error* para indicar que en realidad no tenemos ninguna operación, sino un error. El enumerado quedaría

```
1     type TipoOp is (Error, Suma, Resta, Multiplicacion, Division);
```

Y nuestro procedimiento sería ahora:

```
1     procedure LeerOp(op: out TipoOp) is
2          c: Character;
3     begin
4          Get(c);
5          case c is
6          when '+' =>
7               op := Suma;
8          when '-' =>
9               op := Resta;
10         when '*' =>
11              op := Multiplicacion;
12         when '/' =>
13              op := Division;
14         when others =>
15              op := Error;
16         end case;
17    end;
```

¡Cuidado! Ahora hay que modificar los procedimientos *EvalExpr* y *EscribirOp* puesto que el *TipoOp* tiene un nuevo elemento. En el primer caso podríamos devolver ‘‘`0.0`’’. En el segundo caso podríamos escribir un mensaje indicando que el operador es erróneo.

10.7. Un nuevo prototipo

Por fin podemos leer y evaluar expresiones sencillas. Aquí hay algunas pruebas.

```
; calc
3 + 2
 5.00000E+00
-2
* 2
-4.00000E+00
3 x 2
 0.00000E+00
abc
 0.00000E+00
 0.00000E+00
3.5 / 0
 0.00000E+00
 0.00000E+00
3 / 1
 3.00000E+00
```

Hemos intentado hacer cosas absurdas en la entrada (además de dar entradas correctas) para ver si la calculadora resiste.

Como hemos hecho algunos cambios en el programa repetimos aquí el código tal y como está ahora mismo.

calc.adb

```
1     --
2     -- Calculadora de linea de comandos
3     -- prototipo 1
4     --

6     with Ada.Text_IO;
7     use Ada.Text_IO;

9     procedure Calc is

11          MaxNumArgs: constant Natural := 2;

13          type TipoOp  is (Error, Suma, Resta, Multiplicacion, Division);

15          type TipoArgs is array(1..MaxNumArgs) of Float;

17          -- expresion aritmetica de tipo "3 + 2"
18          type TipoExpr is
19          record
20               op: TipoOp;
21               args: TipoArgs;
22          end record;

24          package FIO is new Float_IO(Float);
25          use FIO;
```

```
27        function EsBlanco(c: Character) return Boolean is
28        begin
29             return c = ' ' or c = ASCII.HT;
30        end;

32        function EsDigito(c: Character) return Boolean is
33        begin
34             return c in '0'..'9';
35        end;

37        procedure SaltarBlancos(eof: in out Boolean) is
38             c: Character;
39             fin: Boolean;
40             eol: Boolean;
41        begin
42             fin := False;
43             while not fin loop
44                  Look_Ahead(c, eol);
45                  if eol then
46                       eof := End_Of_File;
47                       fin := eof;
48                       if not fin then
49                            Skip_Line;
50                       end if;
51                  elsif not EsBlanco(c) then
52                       fin := True;
53                  else
54                       Get(c);
55                  end if;
56             end loop;
57        end;

59        procedure LeerOp(op: out TipoOp) is
60             c: Character;
61        begin
62             Get(c);
63             case c is
64             when '+' =>
65                  op := Suma;
66             when '-' =>
67                  op := Resta;
68             when '*' =>
69                  op := Multiplicacion;
70             when '/' =>
71                  op := Division;
72             when others =>
73                  op := Error;
74             end case;
75        end;
```

```
77          procedure LeerNumero(num: out Float) is
78               c: Character;
79               digito: Integer;
80               fin: Boolean;
81               parteentera: Boolean;
82               fraccion: Float;
83               signo: Float;
84          begin
85               num := 0.0;
86               parteentera := True;
87               fraccion := 1.0;
88               signo := 1.0;

90               Look_Ahead(c, fin);
91               while not fin loop
92                    case c is
93                    when '-' =>
94                         signo := -1.0;
95                         Get(c);
96                    when '+' =>
97                         Get(c);
98                    when '.' =>
99                         parteentera := false;
100                        Get(c);

102                   when '0'..'9' =>
103                        Get(c);
104                        digito := Character'Pos(c) - Character'Pos('0');
105                        if parteentera then
106                             num := num * 10.0 + Float(digito);
107                        else
108                             fraccion := fraccion / 10.0;
109                             num := num + Float(digito) * fraccion;
110                        end if;

112                   when others =>
113                        fin := True;
114                   end case;

116                   if not fin then
117                        Look_Ahead(c, fin);
118                   end if;
119              end loop;
120              num := signo * num;
121         end;
```

```
123         procedure LeerExpr(expr: in out TipoExpr; eof: in out Boolean) is
124         begin
125              SaltarBlancos(eof);
126              if not eof then
127                   LeerNumero(expr.args(1));
128              end if;
129              if not eof then
130                   SaltarBlancos(eof);
131              end if;
132              if not eof then
133                   LeerOp(expr.op);
134              end if;
135              if not eof then
136                   SaltarBlancos(eof);
137              end if;
138              if not eof then
139                   LeerNumero(expr.args(2));
140              end if;
141         end;

143         function NuevaExpr(op: TipoOp; arg1: Float; arg2: Float) return TipoExpr is
144              expr: TipoExpr;
145         begin
146              expr.op := op;
147              expr.args(1) := arg1;
148              expr.args(2) := arg2;
149              return expr;
150         end;

152         procedure EscribirOp(op: TipoOp) is
153         begin
154              case op is
155              when Suma =>
156                   Put("+");
157              when Resta =>
158                   Put("-");
159              when Multiplicacion =>
160                   Put("*");
161              when Division =>
162                   Put("/");
163              when Error =>
164                   Put("???");
165              end case;
166         end;

168         procedure EscribirExpr(expr: TipoExpr) is
169         begin
170              Put(expr.args(1));
171              Put(" ");
172              EscribirOp(expr.op);
173              Put(expr.args(2));
174         end;
```

```
176         function EvalExpr(expr: TipoExpr) return Float is
177              result: Float;
178              arg2: Float;
179         begin
180              case expr.op is
181              when Suma =>
182                   result := expr.args(1) + expr.args(2);
183              when Resta =>
184                   result := expr.args(1) - expr.args(2);
185              when Multiplicacion =>
186                   result := expr.args(1) * expr.args(2);
187              when Division =>
188                   if expr.args(2) = 0.0 then
189                        arg2 := 1.0e-30;
190                   else
191                        arg2 := expr.args(2);
192                   end if;
193                   result := expr.args(1) / arg2;
194              when Error =>
195                   result := 0.0;
196              end case;
197              return result;
198         end;

200         eof: Boolean;
201         expr: TipoExpr;
202         valor: Float;
203    begin
204         eof := False;
205         while not eof loop
206              LeerExpr(expr, eof);
207              if not eof then
208                   valor := EvalExpr(expr);
209                   Put(valor);
210                   New_Line;
211              end if;
212         end loop;
213    end;
—
```

10.8. Segundo asalto

Ahora necesitamos acercarnos más al problema original. Podemos considerar alguna de las cosas que nos quedan y contemplarlas ahora. Por ejemplo, podemos incluir la posibilidad de escribir números romanos. Para hacer esto vamos a tener que hacer varias cosas:

- Necesitamos modificar la lectura de números para que puedan ser romanos.
- Necesitamos calcular el valor real que corresponde a un número romano.

Por lo demás el resto del programa puede quedarse como está. Una vez tenemos nuestra *expr* podemos evaluarla e imprimirla como queramos. Eso es lo bueno de habernos inventado una abstracción.

Un comentario importante: si el programa resulta difícil de cambiar para incluir nuevos elementos (por ejemplo, nuestros números romanos) entonces es que está mal programado. En tal caso haríamos bien en volverlo a empezar desde el principio, intentando hacerlo de un modo un poco más limpio esta vez.

Cuando se hacen cambios en un programa hay que mantener el nivel: no debe romperse la estructura del programa (o los datos) y no se puede forzar el programa para que haga algo para lo que no está preparado. Las chapuzas siempre se pagan (con tiempo de depuración) aunque parecen más fáciles y seductoras (como el lado oscuro de la fuerza). Una vez comienzas a hacer chapuzas... ¡Para siempre dominarán tu destino! (Hasta que cambies de trabajo, por lo menos).

Para empezar, tenemos claro que tenemos que manipular números romanos. Lo único que queremos hacer con ellos es leerlos y calcular su valor. Necesitamos otro tipo de datos:

```
1     MaxLongRom: constant Natural := 50;

3     type TipoDigitoRom is (RomI, RomV, RomX, RomL, RomC, RomD, RomM);

5     type TipoDigitosRom is array(1..MaxLongRom) of TipoDigitoRom;

7     type TipoNumRom is
8     record
9          digitos: TipoDigitosRom;
10         numdigitos: Natural;
11    end record;
```

Dado que no sabemos qué longitud tendrá un número romano hemos optado por utilizar la misma idea que hemos utilizado en el pasado para manipular palabras. Almacenamos un máximo de *MaxLongRom* dígitos junto con el número de dígitos que realmente utilizamos. A partir de ahora, un número romano será un *TipoNumRom* y cualquier cosa que queramos hacer con el la haremos a base de definir operaciones para este tipo.

Vamos a modificar el código suponiendo que tenemos todo lo que podamos desear. La idea es que ahora una expresión podrá ser alguna de las siguientes cosas en la entrada del programa:

```
3 * 2
XI + 7
9 / III
MCM + XXX
```

Sean los números arábigos o romanos, nuestro tipo *TipoExpr* sirve perfectamente para todos los casos. La cuestión es que al leer una expresión puede que alguno o todos los operandos sean números romanos. El código de *LeerExpr* hace en realidad lo que debe: lee un número, un operador y otro número. Luego parece que deberíamos modificar la operación de leer número para que lo lea en un formato u otro. Este es el plan:

- Cambiamos *LeerNumero* por *LeerOperando* en *LeerExpr*. Ahora, *LeerOperando* debe ver si tiene un número romano o arábigo en la entrada y actuar en consecuencia.

Esta es la nueva *LeerExpr*:

```
1     procedure LeerExpr(expr: in out TipoExpr; eof: in out Boolean) is
2     begin
3          SaltarBlancos(eof);
4          if not eof then
5               LeerOperando(expr.args(1));
6          end if;
7          if not eof then
8               SaltarBlancos(eof);
9          end if;
10         if not eof then
11              LeerOp(expr.op);
12         end if;
13         if not eof then
14              SaltarBlancos(eof);
15         end if;
16         if not eof then
17              LeerOperando(expr.args(2));
18         end if;
19    end;
```

Donde *LeerOperando* podría ser justo lo que hemos dicho, si suponemos que hay una función llamada *HayRomano* que nos dice si hay un número romano (o parece haberlo) a continuación en la entrada.

```
1     procedure LeerOperando(num: out Float) is
2     begin
3          if HayRomano then
4               LeerRomano(num);
5          else
6               LeerNumero(num);
7          end if;
8     end;
```

No obstante, tenemos un tipo de datos que se ocupa de manipular números romanos: la llamada a *LeerRomano* tiene mal aspecto. Podría quedar mucho mejor algo como leer un número romano y luego calcular su valor:

```
1     procedure LeerOperando(num: out Float) is
2          rom: TipoNumRom;
3     begin
4          if HayRomano then
5               LeerNumRom(rom);
6               num := ValorNumRom(rom);
7          else
8               LeerNumero(num);
9          end if;
10    end;
```

¿Por qué? Sencillamente por que la llamada que teníamos antes se llamaba *LeerRomano* pero devolvía un objeto de tipo *Float*. ¡Un *Float* no es un número romano! Ignorar este tipo de cosas se termina pagando, por eso preferimos no ignorarlas y mantener el programa limpio en todo momento.

¿Cómo podemos saber si tenemos un número romano en la entrada? Los números romanos han de utilizar alguno de los caracteres romanos. Los arábigos utilizan dígitos arábicos, lo cual es sorprendente. Podemos mirar el siguiente carácter de la entrada y decidir. Como es feo que una función tenga efectos laterales (y mirar en la entrada lo es), optamos por volver a modificar *LeerOperando*:

```
1     procedure LeerOperando(num: out Float) is
2          rom: TipoNumRom;
3          c: Character;
4          fin: Boolean;
5     begin
6          Look_Ahead(c, fin);
7          if EsRomano(c) then
8               LeerNumRom(rom);
9               num := ValorNumRom(rom);
10         else
11              LeerNumero(num);
12         end if;
13    end;
```

Sabemos que *Look_Ahead* no puede encontrar un fin de línea (acabamos de saltar blancos) por lo que podemos ignorar *fin*.

Ver si un carácter es de un número romano es fácil.

```
1     function EsRomano(c: Character) return Boolean is
2          loes: Boolean;
3     begin
4          case c is
5          when 'I' | 'V' | 'X' | 'L' | 'C' | 'D' | 'M' =>
6               loes := True;
7          when others =>
8               loes := False;
9          end case;
10         return loes;
11    end;
```

Ahora tenemos que programar la operación que lee un número romano: *LeerNumRom.* Lo hacemos igual que cuando programamos *LeerPalabra* hace ya algún tiempo. Esta vez hay que detenerse si el siguiente carácter deja de ser válido para un número romano.

```
1     procedure LeerNumRom(rom: out TipoNumRom) is
2          c: Character;
3          fin: Boolean;
4     begin
5          rom.numdigitos := 0;  -- vacio por ahora
6          loop
7               Get(c);
8               rom.numdigitos := rom.numdigitos + 1;
9               rom.digitos(rom.numdigitos) := DigitoRom(c);
10              Look_Ahead(c, fin);
11              if not fin then
12                   fin := not EsRomano(c);
13              end if;
14              exit when fin;
15         end loop;
16    end;
```

Como hicimos bien las cosas, ahora hemos podido utilizar *EsRomano* para ver si un carácter corresponde a un dígito romano. Ya la teníamos. Lo que nos falta es *DigitoRom* para convertir un carácter en un dígito romano.

```
1     function DigitoRom(c: Character) return TipoDigitoRom is
2          d: TipoDigitoRom;
3     begin
4          case c is
5          when 'I' =>
6               d := RomI;
7          when 'V' =>
8               d := RomV;
9          when 'X' =>
10              d := RomX;
11         when 'L' =>
12              d := RomL;
13         when 'C' =>
14              d := RomC;
15         when 'D' =>
16              d := RomD;
17         when others =>
18              d := RomM;
19         end case;
20         return d;
21    end;
```

En teoría, *c* puede ser cualquier carácter. En la práctica hemos comprobado si tenemos un dígito romano antes de leerlo. Para que el *case* cubra todos los casos hemos escrito la última rama del *case* utilizando *others* en lugar de `'M'`.

El valor de un número romano es fácil de calcular si se sabe hacerlo a mano. Inicialmente tenemos cero como valor y tomamos dígito por dígito. Si un dígito es mayor que el que le sigue, entonces se resta su valor del siguiente dígito. En otro caso se suma. Podemos entonces ir acumulando dígitos, pero con signo negativo en los casos en que hay que restar el valor. Esto es lo que nos queda:

```
1     function ValorNumRom(rom: TipoNumRom) return Float is
2          total: Float;
3          valor: Float;
4          sigvalor: Float;
5     begin
6          total := 0.0;
7          for i in 1..rom.numdigitos loop
8               valor := ValorDigitoRom(rom.digitos(i));
9               if i < rom.numdigitos then
10                   sigvalor := ValorDigitoRom(rom.digitos(i+1));
11                   if valor < sigvalor then
12                        valor := - valor;
13                   end if;
14              end if;
15              total := total + valor;
16         end loop;
17         return total;
18    end;
```

Hemos tenido que tener cuidado con el último dígito.

Para calcular el valor de un dígito no vamos a utilizar una función. Como los casos son pocos, es mas sencillo (y más rápido cuando ejecute el programa) utilizar un *array* constante como este:

```
1     ValorDigitoRom: constant array(TipoDigitoRom) of Float :=
2                     (1.0, 5.0, 10.0, 50.0, 100.0, 500.0, 1000.0);
```

En muchas situaciones podemos utilizar tablas o *arrays* para pre-calcular el valor de una función. Ambas cosas son equivalentes. Todo depende de lo que nos resulte más cómodo y de cuántos

valores posibles toma la función como entrada. Si son muchos no es práctico escribir un *array* puesto que éste consumiría una cantidad muy grande de memoria.

¡Y ya lo tenemos!

```
; calc
XIII + 3
 1.60000E+01
```

10.9. Funciones elementales

Debemos también modificar la calculadora para que sea capaz de entender funciones elementales de la forma

```
seno 3
```

o bien

```
coseno XII
```

Vamos a considerar como funciones las siguientes: seno, coseno, tangente y raíz cuadrada. Otras operaciones y funciones se podrían añadir mas tarde de forma análoga.

Si empezamos a pensar cómo hacerlo... tenemos un problema: las expresiones han dejado de ser una operación con dos argumentos. Ahora puede suceder que tengamos una función y un sólo argumento.

Tiene arreglo: podemos hacer que el número de argumentos sea variable (uno o dos) y definir las funciones elementales del mismo modo que hemos definido las operaciones. Considerando esto, tenemos que cambiar nuestro *TipoOp* para que sea como sigue:

```
1     type TipoOp  is (Error, Suma, Resta, Multiplicacion, Division,
2           Seno, Coseno, Tangente, Raiz);
```

En el tipo *TipoExpr* debemos ahora considerar el número de argumentos (que puede ser uno):

```
1     -- expresion aritmetica de tipo "3 + 2" o "seno 43"
2     type TipoExpr is
3     record
4           op: TipoOp;
5           args: TipoArgs;
6           numargs: Natural;
7     end record;
```

Dado que hemos cambiado el tipo de datos, vamos a ajustar las operaciones que tiene (los subprogramas que lo manipulan) antes de hacer nada más. Hay que estar seguros de que, tras el cambio, todas las operaciones hacen lo que deben.

Ya no deberíamos tener una función *NuevaExpr*. Ahora podemos tener expresiones unarias (un argumento) y binarias. Vamos a reemplazar nuestra antigua función por estas dos:

```
1     function NuevaExpr2(op: TipoOp; arg1: Float; arg2: Float) return TipoExpr is
2           expr: TipoExpr;
3     begin
4           expr.op := op;
5           expr.numargs := 2;
6           expr.args(1) := arg1;
7           expr.args(2) := arg2;
8           return expr;
9     end;
```

```
11    function NuevaExpr1(op: TipoOp; arg1: Float) return TipoExpr is
12         expr: TipoExpr;
13    begin
14         expr.op := op;
15         expr.numargs := 1;
16         expr.args(1) := arg1;
17         return expr;
18    end;
```

No es que importe, dado que no se utilizaba la función *NuevaExpr* en el programa. Pero si que importa puesto que ahora no es obvio lo que hace *NuevaExpr*. Es mejor arreglarlo para que si, en el futuro, queremos declarar constantes para expresiones podamos hacerlo sin sorpresas. Lo mismo hay que hacer con *EscribirOp*, por cierto.

La forma de escribir una expresión también ha cambiado. Las expresiones unarias deberían escribirse con el nombre de la operación antes del primer (y único) argumento. Las binarias se escriben como antes.

```
1     procedure EscribirExpr(expr: TipoExpr) is
2     begin
3          if expr.numargs = 1 then
4               EscribirOp(expr.op);
5               Put(" ");
6               Put(expr.args(1));
7          else
8               Put(expr.args(1));
9               Put(" ");
10              EscribirOp(expr.op);
11              Put(expr.args(2));
12         end if;
13    end;
```

La función *EvalExpr* debe ser capaz de evaluar las nuevas expresiones. Basta añadir más casos al *case*:

```
1     when Seno =>
2          result := Sin(expr.args(1));
3     when Coseno =>
4          result := Cos(expr.args(1));
5     when Tangente =>
6          result := Tan(expr.args(1));
7     when Raiz =>
8          result := Sqrt(expr.args(1));
```

Para utilizar funciones elementales, como *Sin*, tenemos que particularizar el paquete de funciones matemáticas elementales para el tipo *Float*. Hay que añadir

```
1     with Ada.Numerics.Generic_Elementary_Functions;
```

como nueva cláusula *use* y, además, declarar

```
1     package GEF is new Ada.Numerics.Generic_Elementary_Functions(Float);
2     use GEF;
```

en la sección de particularización de paquetes.

Ahora tenemos que cambiar la forma en que leemos expresiones. La idea es que nuestro *LeerExpr* es en realidad un procedimiento que lee una expresión binaria. Pero podemos tener también expresiones unarias. Lo que determina si tenemos una u otra es ver si tenemos una función en la entrada o tenemos un número (que puede ser romano o arábigo).

Cambiamos pues *LeerExpr* para que sea:

```
1     procedure LeerExpr(expr: in out TipoExpr; eof: in out Boolean) is
2          c: Character;
3          fin: Boolean;
4     begin
5          SaltarBlancos(eof);
6          if not eof then
7               Look_Ahead(c, fin);
8               if EsFuncion(c) then
9                    LeerExpr1(expr, eof);
10              else
11                   LeerExpr2(expr, eof);
12              end if;
13         end if;
14    end;
```

Podemos implementar *EsFuncion* y olvidarnos de ella.

```
1     function EsFuncion(c: Character) return Boolean is
2     begin
3          return c in 'a'..'z';
4     end;
```

Hemos supuesto que cualquier nombre que empiece por minúscula corresponde a una función (los números romanos los escribimos con mayúsculas siempre).

El procedimiento *LeerExpr2* es el antiguo *LeerExpr*, y lee una expresión binaria. El procedimiento *LeerExpr1* lee una expresión unaria; lo hacemos de un modo similar a *LeerExpr2*:

```
1     procedure LeerExpr1(expr: in out TipoExpr; eof: in out Boolean) is
2     begin
3          LeerOp(expr.op);
4          SaltarBlancos(eof);
5          if not eof then
6               LeerOperando(expr.args(1));
7               expr.numargs := 1;
8          end if;
9     end;
```

Puesto que hemos optado por extender nuestro tipo *TipoOp*, para incluir las funciones como operaciones, parece razonable que tengamos que utilizar *LeerOp* para leer también aquellas operaciones que son funciones.

Ahora una operación puede ser un único signo de operación aritmética o bien una palabra para una función que conocemos. Para no crear casos especiales podríamos leer una palabra y luego mirar de qué operación se trata. Podemos volver a utilizar nuestro antiguo *TipoPalabra* para tal fin. Hemos pues de añadir dicho tipo (véanse problemas anteriores) y la constante *MaxLongPalabra* y el tipo *TipoRangoPalabra* utilizados por el. Además, volvemos a tomar prestado el procedimiento *LeerPalabra* y la función *PalabraEnAda* (que nos será útil para comparar la palabra leída con nuestros nombres de función y operación.

La forma de leer una operación es, como hemos dicho, leer primero una palabra y luego convertirla a una operación.

```
1     procedure LeerOp(op: out TipoOp) is
2          palabra: TipoPalabra;
3     begin
4          LeerPalabra(palabra);
5          if PalabraEnAda(palabra) = "+" then
6               op := Suma;
7          elsif PalabraEnAda(palabra) = "-" then
8               op := Resta;
9          elsif PalabraEnAda(palabra) = "*" then
10              op := Multiplicacion;
11         elsif PalabraEnAda(palabra) = "/" then
12              op := Division;
13         elsif PalabraEnAda(palabra) = NombreSeno then
14              op := Seno;
15         elsif PalabraEnAda(palabra) = NombreCoseno then
16              op := Coseno;
17         elsif PalabraEnAda(palabra) = NombreTangente then
18              op := Tangente;
19         elsif PalabraEnAda(palabra) = NombreRaiz then
20              op := Raiz;
21         else
22              op := Error;
23         end if;
24    end;
```

Puesto que puede ser cuestionable el nombre utilizado para cada una de las funciones, hemos optado por definir constantes al efecto, para que sea trivial cambiarlas si así lo queremos.

```
1     NombreSeno:     constant String := "sen";
2     NombreCoseno:   constant String := "cos";
3     NombreTangente: constant String := "tan";
4     NombreRaiz:     constant String := "raiz";
```

Ahora podemos utilizar nuestra calculadora como se ve a continuación:

```
; calc
sen MCM
 6.15922E-01
3 - XI
-8.00000E+00
raiz X
 3.16228E+00
```

10.10. Memorias

Hemos ido modificando las estructuras de datos que teníamos para que sigan siendo un fiel reflejo de los objetos que manipula el programa; todavía tenemos el programa en un estado razonablemente limpio. Lo mismo hemos hecho con los procedimientos y funciones. No hay ninguno de ellos cuyo nombre mienta, o que haga algo a escondidas sin que su interfaz lo ponga de manifiesto. El resultado de haberlo hecho así es que, en general, resultará fácil hacer cuantas modificaciones sea preciso.

Otra necesidad que teníamos era dotar a la calculadora de memoria. Esa es la modificación que vamos a hacer ahora.

Por un lado tenemos que tener operaciones capaces de recordar el último valor calculado. Esto quiere decir que, además, tenemos que saber cuál es el último valor calculado. Habíamos pensado utilizar ‘‘`M1`’’ y ‘‘`M2`’’ para guardar dicho valor. No obstante, estos nombres nos plantean un problema puesto que la ‘‘`M`’’ podría confundirse con un dígito romano. Cambiamos pues las especificaciones (sí, no sólo se puede hacer sino que en la realidad se hace miles de veces)

para tener una nueva operación que guarde el último valor en la memoria. Podríamos pensar en utilizar

```
mem1
```

y

```
mem2
```

Pero, como podrá verse, es igual de fácil admitir operaciones de la forma

```
mem 1
```

o

```
mem 2
```

que podrían ser capaces de guardar el último valor en la memoria cuyo número se indica como operando. Esta última forma parece más general.

Cuando evaluemos esas expresiones vamos a tener que poder cambiar la memoria de la calculadora, lo que quiere decir que necesitamos una memoria. Por lo demás es cuestión de hacer que cuando un operando sea ‘‘R1’’ o ‘‘R2’’ se utilice la posición correspondiente en la memoria.

Empezaremos por modificar de nuevo nuestro *TipoOp*, para incluir una operación para guardar un valor en la memoria.

```
1     type TipoOp  is (Error, Suma, Resta, Multiplicacion, Division,
2          Seno, Coseno, Tangente, Raiz, Guardar);
```

Ahora tenemos que cambiar el procedimiento capaz de leer una operación, `LeerOp`, para que incluya dicha operación.

```
1      ...
2     elsif PalabraEnAda(palabra) = NombreGuardar then
3          op := Guardar;
4      ...
```

Igual que hicimos con el resto de nombres que ve el usuario, hemos definido una constante para el nombre de la operación:

```
1     NombreGuardar:  constant String := "mem";
```

Naturalmente, hay que cambiar tambien *EscribirOp*. Como el nombre es un nombre de función que empieza por una letra minúscula, todo el código de lectura de expresiones sigue funcionando para la nueva operación. No es preciso cambiar nada, ahora tenemos una nueva función de un argumento.

El programa principal tiene ya una variable *valor*, que mantiene siempre el último valor. Lo que sucede ahora es que, al evaluar una expresión, puede que tengamos que guardar este valor en una memoria. Por lo tanto, necesitamos un tipo de datos para la memoria, una variable para la memoria y suministrar tanto *valor* como la memoria a *EvalExpr* (que ahora los necesita). Declaramos...

```
1     MaxNumMem: constant Natural := 2;
2     type TipoMem is array(1..MaxNumMem) of Float;
```

Y ajustamos un poco el programa principal:

```
1          eof: Boolean;
2          expr: TipoExpr;
3          valor: Float;
4          mem: TipoMem;
```

```
6     begin
7           valor := 0.0;
8           NuevaMem(mem);

10          eof := False;
11          while not eof loop
12                LeerExpr(expr, eof);
13                if not eof then
14                      EvalExpr(expr, valor, mem);
15                      Put(valor);
16                      New_Line;
17                end if;
18          end loop;
19    end;
```

Ahora empezamos inicializando a cero tanto el último valor calculado como la memoria. Además, *EvalExpr* debe ser un procedimiento puesto que debe poder cambiar tanto *valor* como posiblemente *mem.* Su cabecera es ahora

```
1     procedure EvalExpr(expr: TipoExpr;
2                        result: in out Float; mem: in out TipoMem) is
```

Caso de que la operación sea guardar, procedemos de este modo en *EvalExpr*:

```
1     when Guardar =>
2           nummem := Integer(expr.args(1));
3           if nummem in mem'Range then
4                 mem(nummem) := result;
5           end if;
```

Hemos tenido cuidado de que el argumento corresponda a una memoria que exista. Además, como no alteramos *result*, el valor resultante de memorizar un valor es justo ese valor (como suele suceder en las calculadoras).

Está casi todo hecho. Nos falta poder reclamar el valor de cualquier memoria. Habíamos pensado en que ‘‘`R1`’’ fuese el valor de la memoria 1 y ‘‘`R2`’’ el valor de la memoria 2. La situación es muy similar a cuando introdujimos la posibilidad de utilizar números romanos. Ahora hay un tercer tipo de números: nombres de memorias. Dado que nos fue bien en el caso anterior ahora vamos a proceder del mismo modo. Modificamos primero *Leer_Operando* para que (además de números romanos) entienda los ‘‘números de memoria’’. Por cierto, esto requiere que se le suministre a *LeerOperando* el contenido de la memoria.

```
1     procedure LeerOperando(num: out Float; mem: TipoMem) is
2           rom: TipoNumRom;
3           c: Character;
4           fin: Boolean;
5     begin
6           Look_Ahead(c, fin);
7           if EsRomano(c) then
8                 LeerNumRom(rom);
9                 num := ValorNumRom(rom);
10          elsif EsMemoria(c) then
11                LeerNumMem(num, mem);
12          else
13                LeerNumero(num);
14          end if;
15    end;
```

LeerNumMem utiliza la memoria para dejar en *num* el contenido de la memoria cuyo nombre se encuentra en la entrada estándar.

```
1     procedure LeerNumMem(num: out Float; mem: TipoMem) is
2          c: Character;
3          fin: Boolean;
4          digito: Integer;
5     begin
6          Get(c);              -- 'R'
7          Look_Ahead(c, fin);
8          num := 0.0;
9          if not fin then
10              digito := Character'Pos(c) - Character'Pos('0');
11              if digito in mem'Range then
12                   num := mem(digito);
13              end if;
14         end if;
15    end;
```

Si el carácter que sigue a la ''R'' es un número dentro de rango como índice en el array de la memoria, entonces utilizamos dicha posición de la memoria. En cualquier otro caso dejamos que *num* sea cero.

Falta alguna función como *EsMemoria* y, además, es preciso suministrar como argumento *mem* a todos los suprogramas en el camino desde el programa principal hasta *LeerNum*. Pero no falta nada más.

10.11. Y el resultado es...

Tenemos un programa que cumple todos los requisitos que nos habíamos planteado, aunque alguno de los requisitos ha cambiado por el camino. Hay muchas decisiones, completamente arbitrarias, que hemos tomado en el proceso. Desde luego, este no es el mejor programa posible, pero es un comienzo. Ahora podría mejorarse de múltiples formas. Nosotros vamos a dejarlo como está.

calc.adb

```
1     --
2     -- Calculadora de linea de comandos
3     -- prototipo 1
4     --

6     with Ada.Text_IO;
7     use Ada.Text_IO;

9     with Ada.Numerics.Generic_Elementary_Functions;

11    procedure Calc is

13         MaxNumArgs: constant Natural := 2;
14         MaxLongPalabra: constant Integer := 500;
15         MaxLongRom: constant Natural := 50;
16         MaxNumMem: constant Natural := 2;

18         NombreSeno:     constant String := "sen";
19         NombreCoseno:   constant String := "cos";
20         NombreTangente: constant String := "tan";
21         NombreRaiz:     constant String := "raiz";
22         NombreGuardar:  constant String := "mem";

24         subtype TipoRangoPalabra is Integer range 1..MaxLongPalabra;
```

```
26        type TipoOp  is (Error, Suma, Resta, Multiplicacion, Division,
27             Seno, Coseno, Tangente, Raiz, Guardar);

29        type TipoArgs is array(1..MaxNumArgs) of Float;

31        type TipoDigitoRom is (RomI, RomV, RomX, RomL, RomC, RomD, RomM);

33        -- expresion aritmetica de tipo "3 + 2" o "seno 43"
34        type TipoExpr is
35        record
36             op: TipoOp;
37             args: TipoArgs;
38             numargs: Natural;
39        end record;

41        type TipoMem is array(1..MaxNumMem) of Float;

43        type TipoDigitosRom is array(1..MaxLongRom) of TipoDigitoRom;

45        type TipoNumRom is
46        record
47             digitos: TipoDigitosRom;
48             numdigitos: Natural;
49        end record;

51        type TipoPalabra is
52        record
53             letras: String(TipoRangoPalabra);
54             long: Natural;
55        end record;

57        ValorDigitoRom: constant array(TipoDigitoRom) of Float :=
58                        (1.0, 5.0, 10.0, 50.0, 100.0, 500.0, 1000.0);

60        package FIO is new Float_IO(Float);
61        use FIO;

63        package GEF is new Ada.Numerics.Generic_Elementary_Functions(Float);
64        use GEF;

66        function EsBlanco(c: Character) return Boolean is
67        begin
68             return c = ' ' or c = ASCII.HT;
69        end;

71        function EsRomano(c: Character) return Boolean is
72             loes: Boolean;
73        begin
74             case c is
75             when 'I' | 'V' | 'X' | 'L' | 'C' | 'D' | 'M' =>
76                  loes := True;
77             when others =>
78                  loes := False;
79             end case;
80             return loes;
81        end;
```

```
83          function EsFuncion(c: Character) return Boolean is
84          begin
85               return c in 'a'..'z';
86          end;

88          function EsMemoria(c: Character) return Boolean is
89          begin
90               return c = 'R';
91          end;

93          procedure SaltarBlancos(eof: in out Boolean) is
94               c: Character;
95               fin: Boolean;
96               eol: Boolean;
97          begin
98               fin := False;
99               while not fin loop
100                   Look_Ahead(c, eol);
101                   if eol then
102                        eof := End_Of_File;
103                        fin := eof;
104                        if not fin then
105                             Skip_Line;
106                        end if;
107                   elsif not EsBlanco(c) then
108                        fin := True;
109                   else
110                        Get(c);
111                   end if;
112              end loop;
113         end;

115         procedure LeerPalabra(palabra: in out TipoPalabra) is
116              c: Character;
117              fin: Boolean;
118         begin
119              palabra.long := 0;   -- vacia por ahora
120              loop
121                   Get(c);
122                   palabra.long := palabra.long + 1;
123                   palabra.letras(palabra.long) := c;
124                   Look_Ahead(c, fin);
125                   if not fin then
126                        fin := EsBlanco(c);
127                   end if;
128                   exit when fin;
129              end loop;
130         end;

132         function PalabraEnAda(palabra: TipoPalabra) return String is
133         begin
134              return palabra.letras(1..palabra.long);
135         end;
```

```
137         procedure LeerOp(op: out TipoOp) is
138              palabra: TipoPalabra;
139         begin
140              LeerPalabra(palabra);
141              if PalabraEnAda(palabra) = "+" then
142                   op := Suma;
143              elsif PalabraEnAda(palabra) = "-" then
144                   op := Resta;
145              elsif PalabraEnAda(palabra) = "*" then
146                   op := Multiplicacion;
147              elsif PalabraEnAda(palabra) = "/" then
148                   op := Division;
149              elsif PalabraEnAda(palabra) = NombreSeno then
150                   op := Seno;
151              elsif PalabraEnAda(palabra) = NombreCoseno then
152                   op := Coseno;
153              elsif PalabraEnAda(palabra) = NombreTangente then
154                   op := Tangente;
155              elsif PalabraEnAda(palabra) = NombreRaiz then
156                   op := Raiz;
157              elsif PalabraEnAda(palabra) = NombreGuardar then
158                   op := Guardar;
159              else
160                   op := Error;
161              end if;
162         end;

164         procedure LeerNumero(num: out Float) is
165              c: Character;
166              digito: Integer;
167              fin: Boolean;
168              parteentera: Boolean;
169              fraccion: Float;
170              signo: Float;
171         begin
172              num := 0.0;
173              parteentera := True;
174              fraccion := 1.0;
175              signo := 1.0;

177              Look_Ahead(c, fin);
178              while not fin loop
179                   case c is
180                   when '-' =>
181                        signo := -1.0;
182                        Get(c);
183                   when '+' =>
184                        Get(c);
185                   when '.' =>
186                        parteentera := false;
187                        Get(c);
```

```
189                 when '0'..'9' =>
190                      Get(c);
191                      digito := Character'Pos(c) - Character'Pos('0');
192                      if parteentera then
193                           num := num * 10.0 + Float(digito);
194                      else
195                           fraccion := fraccion / 10.0;
196                           num := num + Float(digito) * fraccion;
197                      end if;

199                 when others =>
200                      fin := True;
201                 end case;

203                 if not fin then
204                      Look_Ahead(c, fin);
205                 end if;
206            end loop;
207            num := signo * num;
208       end;

210       function DigitoRom(c: Character) return TipoDigitoRom is
211            d: TipoDigitoRom;
212       begin
213            case c is
214            when 'I' =>
215                 d := RomI;
216            when 'V' =>
217                 d := RomV;
218            when 'X' =>
219                 d := RomX;
220            when 'L' =>
221                 d := RomL;
222            when 'C' =>
223                 d := RomC;
224            when 'D' =>
225                 d := RomD;
226            when others =>
227                 d := RomM;
228            end case;
229            return d;
230       end;
```

```
232         procedure LeerNumRom(rom: out TipoNumRom) is
233              c: Character;
234              fin: Boolean;
235         begin
236              rom.numdigitos := 0;  -- vacio por ahora
237              loop
238                   Get(c);
239                   rom.numdigitos := rom.numdigitos + 1;
240                   rom.digitos(rom.numdigitos) := DigitoRom(c);
241                   Look_Ahead(c, fin);
242                   if not fin then
243                        fin := not EsRomano(c);
244                   end if;
245                   exit when fin;
246              end loop;
247         end;

249         function ValorNumRom(rom: TipoNumRom) return Float is
250              total: Float;
251              valor: Float;
252              sigvalor: Float;
253         begin
254              total := 0.0;
255              for i in 1..rom.numdigitos loop
256                   valor := ValorDigitoRom(rom.digitos(i));
257                   if i < rom.numdigitos then
258                        sigvalor := ValorDigitoRom(rom.digitos(i+1));
259                        if valor < sigvalor then
260                             valor := - valor;
261                        end if;
262                   end if;
263                   total := total + valor;
264              end loop;
265              return total;
266         end;

268         procedure LeerNumMem(num: out Float; mem: TipoMem) is
269              c: Character;
270              fin: Boolean;
271              digito: Integer;
272         begin
273              Get(c);    -- 'R'
274              Look_Ahead(c, fin);
275              num := 0.0;
276              if not fin then
277                   digito := Character'Pos(c) - Character'Pos('0');
278                   if digito in mem'Range then
279                        num := mem(digito);
280                   end if;
281              end if;
282         end;
```

```
284         procedure LeerOperando(num: out Float; mem: TipoMem) is
285              rom: TipoNumRom;
286              c: Character;
287              fin: Boolean;
288         begin
289              Look_Ahead(c, fin);
290              if EsRomano(c) then
291                   LeerNumRom(rom);
292                   num := ValorNumRom(rom);
293              elsif EsMemoria(c) then
294                   LeerNumMem(num, mem);
295              else
296                   LeerNumero(num);
297              end if;
298         end;

300         procedure LeerExpr2(expr: in out TipoExpr;
301                             mem: TipoMem; eof: in out Boolean) is
302         begin
303              LeerOperando(expr.args(1), mem);
304              if not eof then
305                   SaltarBlancos(eof);
306              end if;
307              if not eof then
308                   LeerOp(expr.op);
309              end if;
310              if not eof then
311                   SaltarBlancos(eof);
312              end if;
313              if not eof then
314                   LeerOperando(expr.args(2), mem);
315                   expr.numargs := 2;
316              end if;
317         end;

319         procedure LeerExpr1(expr: in out TipoExpr;
320                             mem: TipoMem; eof: in out Boolean) is
321         begin
322              LeerOp(expr.op);
323              SaltarBlancos(eof);
324              if not eof then
325                   LeerOperando(expr.args(1), mem);
326                   expr.numargs := 1;
327              end if;
328         end;
```

```
330         procedure LeerExpr(expr: in out TipoExpr;
331                            mem: TipoMem; eof: in out Boolean) is
332              c: Character;
333              fin: Boolean;
334         begin
335              SaltarBlancos(eof);
336              if not eof then
337                   Look_Ahead(c, fin);
338                   if EsFuncion(c) then
339                        LeerExpr1(expr, mem, eof);
340                   else
341                        LeerExpr2(expr, mem, eof);
342                   end if;
343              end if;
344         end;

346         procedure EscribirOp(op: TipoOp) is
347         begin
348              case op is
349              when Suma =>
350                   Put("+");
351              when Resta =>
352                   Put("-");
353              when Multiplicacion =>
354                   Put("*");
355              when Division =>
356                   Put("/");
357              when Seno =>
358                   Put(NombreSeno);
359              when Coseno =>
360                   Put(NombreCoseno);
361              when Tangente =>
362                   Put(NombreTangente);
363              when Raiz =>
364                   Put(NombreRaiz);
365              when Guardar =>
366                   Put(NombreGuardar);
367              when Error =>
368                   Put("???");
369              end case;
370         end;

372         function NuevaExpr2(op: TipoOp; arg1: Float; arg2: Float) return TipoExpr is
373              expr: TipoExpr;
374         begin
375              expr.op := op;
376              expr.numargs := 2;
377              expr.args(1) := arg1;
378              expr.args(2) := arg2;
379              return expr;
380         end;
```

```
382         function NuevaExpr1(op: TipoOp; arg1: Float) return TipoExpr is
383              expr: TipoExpr;
384         begin
385              expr.op := op;
386              expr.numargs := 1;
387              expr.args(1) := arg1;
388              return expr;
389         end;

391         procedure EscribirExpr(expr: TipoExpr) is
392         begin
393              if expr.numargs = 1 then
394                   EscribirOp(expr.op);
395                   Put(" ");
396                   Put(expr.args(1));
397              else
398                   Put(expr.args(1));
399                   Put(" ");
400                   EscribirOp(expr.op);
401                   Put(expr.args(2));
402              end if;
403         end;

405         procedure EvalExpr(expr: TipoExpr;
406                         result: in out Float; mem: in out TipoMem) is
407              arg2: Float;
408              nummem: Integer;
409         begin
410              case expr.op is
411              when Suma =>
412                   result := expr.args(1) + expr.args(2);
413              when Resta =>
414                   result := expr.args(1) - expr.args(2);
415              when Multiplicacion =>
416                   result := expr.args(1) * expr.args(2);
417              when Division =>
418                   if expr.args(2) = 0.0 then
419                        arg2 := 1.0e-30;
420                   else
421                        arg2 := expr.args(2);
422                   end if;
423                   result := expr.args(1) / arg2;
```

```
425             when Seno =>
426                  result := Sin(expr.args(1));
427             when Coseno =>
428                  result := Cos(expr.args(1));
429             when Tangente =>
430                  result := Tan(expr.args(1));
431             when Raiz =>
432                  result := Sqrt(expr.args(1));
433             when Guardar =>
434                  nummem := Integer(expr.args(1));
435                  if nummem in mem'Range then
436                       mem(nummem) := result;
437                  end if;
438             when Error =>
439                  result := 0.0;
440             end case;
441        end;

443        procedure NuevaMem(mem: out TipoMem) is
444        begin
445             for i in mem'Range loop
446                  mem(i) := 0.0;
447             end loop;
448        end;

450        eof: Boolean;
451        expr: TipoExpr;
452        valor: Float;
453        mem: TipoMem;

455   begin
456        valor := 0.0;
457        NuevaMem(mem);

459        eof := False;
460        while not eof loop
461             LeerExpr(expr, mem, eof);
462             if not eof then
463                  EvalExpr(expr, valor, mem);
464                  Put(valor);
465                  New_Line;
466             end if;
467        end loop;
468   end;
—
```

Problemas

1 Cambia la forma de hacer que se reclame un valor de la memoria de tal forma que no sea preciso hacer que la lectura de operandos acceda a la memoria. Haz que sólo *EvalExpr* necesite utilizar *mem.*

2 Haz que la calculadora escriba mensajes apropiados si hay expresiones erróneas en la entrada del programa.

3 Haz que la calculadora sea capaz de dibujar funciones matemáticas simples.

4 Dota a la calculadora de modo hexadecimal y octal, de tal forma que sea posible hacer aritmética con números en base 16 y base 8 y solicitar cambios de base.

11 — Estructuras dinámicas

11.1. Tipos de memoria

Las variables (y constantes) que hemos estado utilizando hasta el momento son de uno de estos dos tipos:

1 **Variables estáticas o globales**. Estas viven en la memoria durante todo el tiempo que dura la ejecución del programa. Las variables del programa principal son de este tipo y, si hubiésemos permitido declarar las variables antes que los subprogramas, entonces todos los subprogramas habrían podido verlas y utilizarlas.

2 **Variables automáticas o locales**. Estas viven en la pila de memoria que se utiliza para controlar las llamadas a procedimiento. Cuando se produce la invocación de un procedimiento se añade a la cima de la pila espacio para las variables locales del procedimiento al que se invoca (además de para los parámetros y para otras cosas). Cuando la llamada a procedimiento termina se libera ese espacio de la cima de la pila. Dicho de otro modo, estas variables se crean y destruyen automáticamente cuando se invocan procedimientos y cuando las invocaciones terminan.

Disponer sólo de estos dos tipos de almacenamiento o variables hace que resulte necesario saber qué tamaño van a tener los objetos que va a manipular el programa antes de ejecutarlo. Por ejemplo, si vamos a manipular palabras, tenemos que fijar un límite para la palabra más larga que queramos manipular y utilizar un vector de caracteres de ese tamaño para todas y cada una de las palabras que use nuestro programa. Esto tiene el problema de que, por un lado, no podemos manejar palabras que superen ese límite y, por otro, gastamos memoria del ordenador de forma innecesaria para palabras mas pequeñas.

Además, por el momento resulta imposible crear variables durante la ejecución del programa, de tal forma que sobrevivan a llamadas a procedimiento.

Hay una solución para estos problemas. Para poder manipular **objetos de tamaño variable** y para poder **crear objetos nuevos** disponemos en todos los lenguajes de programación de un tercer tipo de variable:

3 **variables dinámicas**. Estas variables pueden crearse mediante una sentencia cuando es preciso y pueden destruirse cuando dejan de ser necesarias. A la primera operación se la suele denominar **asignación de memoria dinámica** y la segunda **liberación de memoria dinámica**.

11.2. Variables dinámicas

Para nosotros, el nombre de una variable (local o global) es lo mismo que la memoria utilizada por dicha variable. Una variable es un nombre para un valor guardado en la memoria del ordenador. Esto está bien para variables locales y globales. Las variables dinámicas funcionan de otro modo: **las variables dinámicas no tienen nombre**.

Una variable dinámica es un nuevo trozo de memoria que podemos pedir durante la ejecución del programa (por eso se denomina ‘‘dinámica’’). Este trozo de la memoria no corresponde a la declaración de ninguna variable de las que tenemos en el programa. Del mismo modo que podemos pedir un nuevo trozo de memoria también podemos devolverlo (allí a dónde lo hemos pedido) cuando deje de ser útil.

Consideremos un ejemplo de uso. Con variables estáticas podemos declarar un *string* de un tamaño dado en el momento de la declaración. Su tamaño podría variar en distintas ejecuciones y ser 100 o 200, pero, una vez alcanzamos la declaración, siempre tendrá el mismo tamaño. Con

variables dinámicas la idea es que durante la ejecución del programa podemos ver cuánta memoria necesitamos para almacenar nuestro *string* y pedir justo esa nueva cantidad de memoria para un nuevo *string*. Cuando el *string* deja de sernos útil podemos devolver al sistema la memoria que hemos pedido, para que pueda utilizarse para otros propósitos. Piensa que en el primer caso (estático) el *string* siempre tiene el mismo tamaño; en el segundo tendrá uno u otro en función de cuánta memoria pidamos. Piensa también que en el primer caso el *string* existe durante toda la vida del procedimiento donde está declarado; en el segundo sólo existe cuando pidamos su memoria, y podría sobrevivir a cuando el procedimiento que lo ha creado termine.

Usar variables dinámicas es fácil si se entienden bien los tres párrafos anteriores. Te sugerimos que los vuelvas a leer después, cuando veamos algunos ejemplos.

11.3. Punteros

Las variables dinámicas se utilizan mediante tipos de datos que son capaces de actuar como ‘‘flechas’’ o ‘‘apuntadores’’ y señalar o apuntar a cualquier posición en la memoria. A estos tipos se los denomina **punteros**. Un puntero es tan sólo una variable que guarda una dirección de memoria. Aunque esto sencillo, suele resultar confuso la primera vez que se ve, y requiere algo de esfuerzo para hacerse con ello.

Si tenemos un tipo de datos podemos construir un nuevo tipo que sea un puntero a dicho tipo de datos. Por ejemplo, si tenemos el tipo de datos *Integer*, podemos declarar un tipo de datos que sea ‘‘puntero a *Integer*’’. Un puntero a *Integer* podrá apuntar a cualquier zona de la memoria en la que tengamos un *Integer*. La idea entonces es que, si en tiempo de ejecución queremos un *nuevo* entero, podemos pedir nueva memoria para un *Integer* y utilizar un puntero a *Integer* para referirnos a ella.

Con un ejemplo se verá todo mas claro. En Ada se declaran tipos de datos puntero con las palabras reservadas *access all*. Por ejemplo, esto declara un nuevo tipo de datos para representar punteros a enteros:

```
type TipoPtrEntero is access all Integer;
```

Igualmente, esto declara un tipo de datos para punteros a datos de tipo *TipoPalabra*:

```
type TipoPtrPalabra is access all TipoPalabra;
```

Las variables de tipo *TipoPtrEntero* pueden o bien **apuntar** a una variable de tipo *Integer* o bien tener el valor especial **null**. Este es un valor nulo que representa que el puntero no apunta a ningún sitio. También se le suele llamar *Nil*.

Si tenemos un tipo para punteros a entero, *TipoPtrEntero*, podemos declarar una variable de este tipo como de costumbre:

```
pentero: TipoPtrEntero;
```

Esto tiene como efecto lo que podemos ver en la figura 11.1.

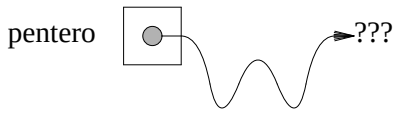


Figura 11.1: *Puntero recién declarado. No sabemos dónde apunta.*

Tenemos una variable de tipo puntero a entero, esto es, capaz de referirse o apuntar a enteros, pero no le hemos dado un valor inicial y no sabemos hacia donde apunta (qué dirección de memoria contiene la variable). Fíjate bien en que *no tenemos ningún entero*. Tenemos algo capaz de referirse a enteros, pero no tenemos entero alguno.

Antes de hacer nada podemos inicializar dicha variable asignando el literal *null*, que hace que el puntero no apunte a ningún sitio.

```
pentero := null;
```

El efecto de esto en el puntero lo podemos representar como muestra la figura 11.2 (utilizamos una simbología similar a poner una línea eléctrica a tierra para indicar que el puntero no apunta a ningún sitio, tal y como se hace habitualmente).

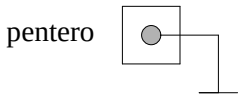


Figura 11.2: *Ejemplo de puntero a nil. No apunta a ningún sitio.*

Pues bien, ¡Llegó el momento! Por primera vez vamos a crear nosotros una nueva variable (sin nombre alguno, eso sí). Hasta el momento siempre lo había hecho Ada por nosotros. Pero ahora, teniendo un puntero a entero, podemos crear un nuevo entero y utilizar el puntero para referirnos a el. Esta sentencia

```
pentero := new Integer;
```

crea una nueva variable de tipo *Integer* (sin nombre alguno) y hace que su dirección se le asigne a *pentero*. El efecto es el que muestra la figura 11.3.

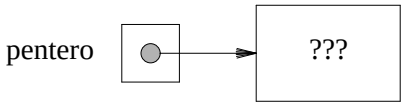


Figura 11.3: *Un puntero apuntando a un nuevo entero*

Ahora el puntero apunta a un nuevo entero. Eso sí, como no hemos inicializado el entero no sabemos qué valor tiene el entero.

Nótese que *tenemos dos variables* en este momento. Una con nombre, el puntero, y una sin nombre, el entero. En realidad del puntero sólo nos interesa que nos sirve para utilizar el entero. Pero una cosa es el entero y otra muy distinta es el puntero.

En Ada, para referirnos a la variable a la que apunta un puntero tenemos que añadir ‘‘`.all`’’ tras el nombre del puntero (¡Pero no estamos utilizando ningún *record*! Es sólo que la sintaxis en Ada es así). Esta sentencia inicializa el entero para que su valor sea 4:

```
pentero.all := 4;
```

El resultado puede verse en la figura 11.4.

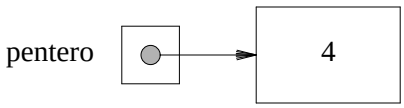


Figura 11.4: *Un puntero que apunta a nuestro entero ya inicializado.*

Nótese que esta asignación ha cambiado el entero. El puntero sigue teniendo el mismo valor, esto es, la dirección del entero al que apunta.

Al acto de referirse al elemento al que apunta un puntero se le denomina vulgarmente *atravesar el puntero* y más técnicamente **aplicar una indirección** al puntero. De hecho, se suele decir que el puntero es en realidad una indirección para indicar que no es directamente el dato que

nos interesa (y que hay que efectuar una indirección para obtenerlo).

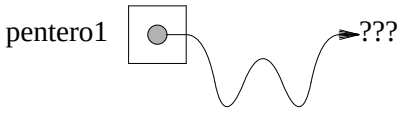
Es un error atravesar un puntero que no está apuntando a un elemento. Por ejemplo, utilizar *pentero.all* cuando el valor de *pentero* es *null* provoca la detención del programa con una indicación de error. Utilizar un puntero cuando no está inicializado es aún peor: no sabemos dónde apunta el puntero y podemos estar accediendo (¡al azar!) a cualquier parte de la memoria. El resultado es muy parecido a un poltergueist.

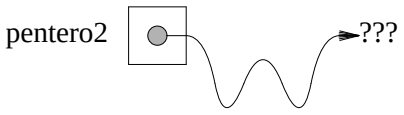
Respecto al estilo, los nombres de tipos puntero deberían ser siempre identificadores que comiencen por *TipoPtr* para indicar que se trata de un tipo de puntero. Igualmente, los nombres de variables de tipo puntero deberían dejar claro que son un puntero; por ejemplo pueden ser siempre identificadores cuyo nombre comience por *p*. Esto permite ver rápidamente cuándo tenemos un puntero entre manos y cuando no, lo que evita muchos errores y sufrimiento innecesario.

11.4. Juegos con punteros

Vamos a ver como ejemplo algunas declaraciones y sentencias que manipulan varios punteros. Tras cada una mostramos el resultado de su ejecución. Empecemos por declarar dos punteros:

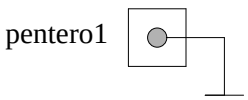
```
pentero1: TipoPtrEntero;
pentero2: TipoPtrEntero;
```

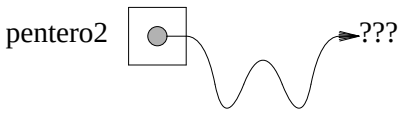




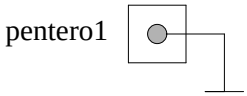
Y ahora ejecutemos algunas sentencias...

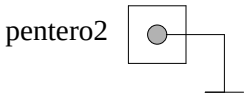
```
pentero1 := null;
```



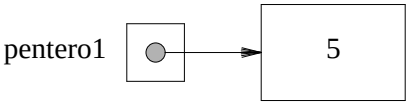


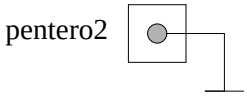

```
pentero2 := pentero1;
```



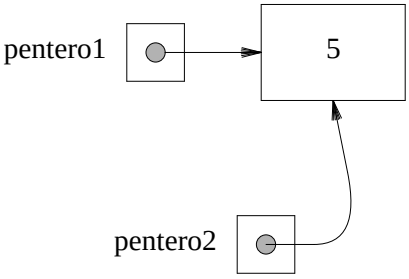



```
pentero1 := new Integer'(5);
```

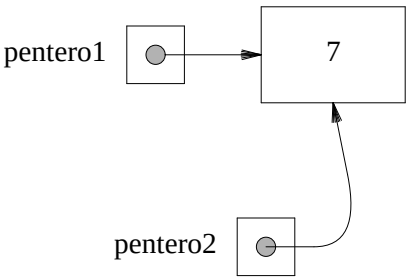





```
pentero2 := pentero1;
```




```
pentero2.all := 7;
```



Si en este punto ejecutamos

```
Put(pentero1.all);
```

veríamos que el valor de *pentero1.all* es 7. Habrás visto que esta sentencia

```
puntero1.all := puntero2.all;
```

copia el valor al que apunta *puntero2* a la variable a la que apunta *puntero1*, pero no modifica ningún puntero. En cambio

```
puntero1 := puntero2;
```

modifica *puntero1* y lo hace apuntar allí donde apunte *puntero2*.

La clave para entender cómo utilizar los punteros es distinguir muy claramente entre el puntero y el objeto al que apuntan. Se recomienda dibujar todas las operaciones con punteros que se programen, tal y como hemos estado haciendo aquí, hasta familiarizarse como los mismos.

11.5. Devolver la memoria al olvido

Cuando deja de ser necesaria la memoria que se ha pedido hay que liberarla. Esto es, de igual modo que ejecutamos

```
pentero := new Integer;
```

cuando queremos crear un nuevo entero, tenemos que destruir dicho entero cuando ya no nos resulte útil. La forma de hacer esto en Ada es utilizar el procedimiento *Ada.Unchecked_Deallocation*. Este procedimiento es un procedimiento general que hay que instanciar para el tipo de datos que queremos liberar y para el tipo de puntero que utilizamos. Una vez instanciado, podemos pasarle un puntero para liberar la memoria a la que apunta el mismo. *En este curso a este procedimiento siempre lo llamaremos Delete.*

Por ejemplo, este programa crea una variable dinámica para guardar un entero en ella, entonces inicializa la variable a 3, imprime el contenido de la variable y, por último, libera la memoria dinámica solicitada anteriormente.

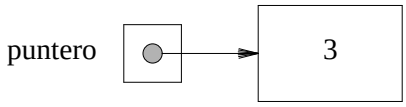
```
1     with Ada.Text_IO;
2     use Ada.Text_IO;

4     with Ada.Unchecked_Deallocation;

6     procedure Ptr is

8           type TipoPtrEntero is access all Integer;

10          -- Creamos un procedure llamado "Delete" para liberar
11          -- la memoria dinamica a la que apunta un TipoPtrEntero, utilizada
12          -- para guardar un Integer.
13          procedure Delete is new Ada.Unchecked_Deallocation(Integer,TipoPtrEntero);

15          package Integer_InOut is new Integer_IO(Integer);
16          use Integer_InOut;

18          puntero: TipoPtrEntero;
19    begin
20          puntero := null;          -- solo como buena costumbre.

22          puntero := new Integer;
23          puntero.all := 3;
24          Put(puntero.all);
```

```
26          -- Liberamos la memoria del entero al que apunta puntero
27          -- ya que no la utilizamos mas
28          Delete(puntero);
29          puntero := null;           -- de nuevo buenas costumbres.
30      end;
```

Las declaraciones

```
4       with Ada.Unchecked_Deallocation;
 ...
13          procedure Delete is new Ada.Unchecked_Deallocation(Integer,TipoPtrEntero);
```

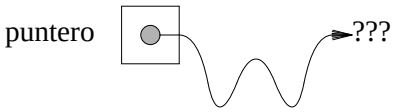
han particularizado un procedimiento llamado *Delete* para liberar elementos de tipo *Integer* alojados en memoria dinámica. Antes de llamar a *Delete* la situación de nuestra memoria podría ser esta:



La llamada

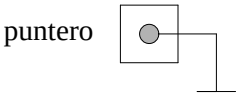
```
Delete(puntero)
```

libera la memoria a la que apunta el puntero. *A partir de este momento no puede utilizarse el puntero*. Téngase en cuenta que la memoria a la que apuntaba ahora podría utilizarse para cualquier otra cosa, dado que hemos dicho que ya no la queremos utilizar más para guardar el elemento al que apuntábamos. En este momento la situación ha pasado a ser:



por lo que es más seguro hacer que el puntero no apunte a ningún sitio a partir de este momento, para evitar atravesarlo por error:

```
puntero := null;
```



Aplicar una indirección a un puntero (esto es, utilizar el valor al que apunta) cuando se ha liberado la memoria a la que apuntaba es un error de consecuencias catastróficas.

Normalmente no hay forma de detectar el error en tiempo de ejecución y lo que sucede es que, si modificamos la variable a la que apunta el puntero, alguna otra variable cambia de modo mágico (¡De magia nada! la memoria a la que apuntaba el puntero ahora la está utilizando otra variable, y al modificarla hemos cambiado otra variable). En ocasiones el lenguaje y/o el sistema operativo pueden detectar que el elemento al otro lado del puntero o no se encuentra en una zona de memoria válida o no tiene un valor dentro del rango para el tipo de datos al que corresponde y se producirá un error en tiempo de ejecución. Pero en la mayoría de los casos, *utilizar memoria ya liberada es un poltergueist*.

Como depurar estos errores cuando se comenten es extremadamente difícil, se suele intentar evitarlos a base de buenas costumbres y mucho cuidado. Por ejemplo, asignando *null* inmediatamente después de liberar un puntero para que el sistema pueda detectar si intentamos atravesarlo y detener la ejecución del programa, en lugar de dejarlo continuar y alterar memoria que no debería alterar.

Hemos de decir que somos malvados. En Ada, el lenguaje inicializa casi todos los punteros (aquellos que no son parámetros) a *null* y también se encarga de liberar la memoria dinámica cuando no quedan punteros apuntando hacia ella, sin que tengamos que llamar a *Delete* o ninguna otra función. A esta última habilidad se la llama **recolección de basura**.

Sucede que muchos lenguajes notables, por ejemplo Pascal, C y C++, no tienen esta habilidad. Es muy probable (¡es seguro!) que tengas que utilizar un lenguaje de programación que ni libere la memoria automáticamente ni se ocupe de inicializar algunos de tus punteros si no lo haces tú. Por esto, en este curso, *hay que liberar la memoria* manualmente en cuanto deje de ser útil, tal y como se ha visto anteriormente. Y por la misma razón, *hay que inicializar los punteros* y no puede suponerse que el lenguaje los inicializa. Ahora puede resultar molesto seguir estas normas, pero te ahorrarán muchas noches depurando.

11.6. Punteros a registros

Antes de completar nuestra discusión de punteros hay que mencionar que Ada dispone de una ayuda sintáctica para utilizar punteros cuando se refieren a *records*. Por ejemplo, dadas la siguientes declaraciones:

```
1     type TipoNodo is
2     record
3          valor: Integer;
4          peso: Integer;
5     end record;

7     type TipoPtrNodo is access all TipoNodo;


10    pnodo: TipoPTrNodo
```

Y supuesto que hemos ejecutado

```
pnodo := new TipoNodo;
```

No es preciso escribir

```
pnodo.all.valor
```

para referirnos al campo *valor* del nodo al que apunta *pnodo*. En su lugar, se escribe

```
pnodo.valor
```

y Ada se ocupa de atravesar automáticamente el puntero para acceder al campo *valor*. Pero hay que recordar que estamos atravesando un puntero. Esto resulta fácil si hemos utilizado un nombre (como *pnodo*) que deja claro que tenemos un puntero entre las manos.

11.7. Listas enlazadas

Un buen ejemplo de cómo utilizar punteros es ver la implementación de una estructura de datos que se utiliza siempre que se quieren tener **colecciones de datos de tamaño variable**. Esto es, algo que (igual que un *array*) permite tener una secuencia ordenada de elementos pero que (al contrario que un *array*) puede crecer de tamaño cuando son precisos más elementos en la secuencia.

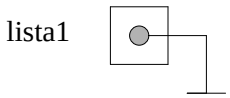
Intenta prestar atención a cómo se utilizan los punteros. Cómo sea una lista o una cola o una pila (aunque son cosas que te harán falta) es algo de lo que aún no deberías preocuparte. Una vez domines la técnica de la programación podrás aprender más sobre algoritmos y sobre estructuras de datos. Pero hay que aprender a andar antes de empezar a correr.

Una lista se define como algo que es una de estas dos cosas:

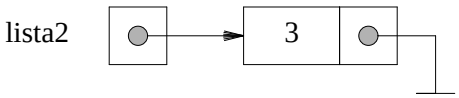
1 Una lista vacía. Esto es, nada.

2 Un elemento de un tipo de datos determinado y el resto de la lista (que es también una lista). A esta pareja se la suele llamar **nodo** de la lista.

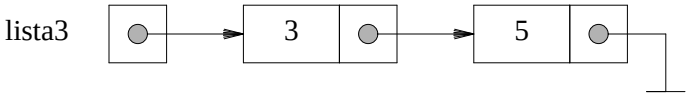
Supongamos que tenemos una lista de enteros. Esto sería una lista vacía:



Y esto una lista con un elemento (esto es, con un nodo que contiene el número 3):



Igualmente, esto es una lista con dos elementos, el 3 y el 5:



El tipo de datos en Ada para una lista podría ser como sigue:

```
1     type TipoNodoEntero;

3     type TipoListaEnteros is access all TipoNodoEntero;

5     type TipoNodoEntero is
6     record
7          valor: Integer;
8          siguiente: TipoListaEnteros;
9     end record;
```

Una lista es un puntero a un nodo. Un nodo contiene un valor y el resto de la lista, que a su vez es una lista. El valor lo declaramos como un campo del registro y el puntero al siguiente elemento de la lista (o el resto de la lista) como un campo del tipo lista.

Dado que cada tipo de datos (lista y nodo) hace referencia al otro ha sido preciso declarar

```
type TipoNodoEntero;
```

antes de declarar *TipoListaEnteros*, para romper la circularidad. Esto se conoce como una declaración incompleta, que sirve a Ada de adelanto para que nos deje utilizar *TipoNodoEntero* en otras declaraciones que siguen. Posteriormente hay que declarar dicho tipo, claro está.

Crear una lista vacía es tan simple como inicializar a *null* una variable de tipo *TipoListaEnteros.* Esta función hace el trabajo.

```
1     function ListaEnterosVacia return TipoListaEnteros is
2          l: TipoListaEnteros;
3     begin
4          l := null;
5          return l;
6     end;
```

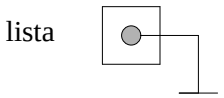
Si tenemos una variable

```
lista: TipoListaEnteros;
```

y la inicializamos así

```
lista := ListaEnterosVacia;
```

entonces esta variable quedará...



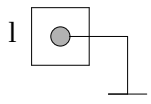
Para ver si una lista es vacía podemos implementar una función como esta:

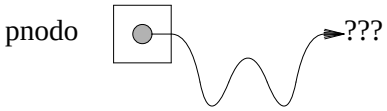
```
1     function EsListaEnterosVacia(l: TipoListaEnteros) return Boolean is
2     begin
3          return l = null;
4     end;
```

Insertar y extraer elementos de una lista suele hacerse de un modo u otro según se quiera utilizar la lista como una pila o como una cola. Si lo que se quiere es una pila, también conocida como **LIFO** (o *Last In is First Out*, el último en entrar es el primero en salir) entonces se suelen insertar los elementos por el comienzo de la lista (la parte izquierda de la lista según la dibujamos) y se suelen extraer elementos también del mismo extremo de la lista. Vamos a hacer esto en primer lugar. Este procedimiento inserta un nodo al principio de la lista.

```
1     procedure InsertarEntero(l: in out TipoListaEnteros; n: Integer) is
2          pnodo: TipoListaEnteros;
3     begin
4          pnodo := new TipoNodoEntero;
5          pnodo.valor := n;
6          pnodo.siguiente := l;
7          l := pnodo;
8     end;
```

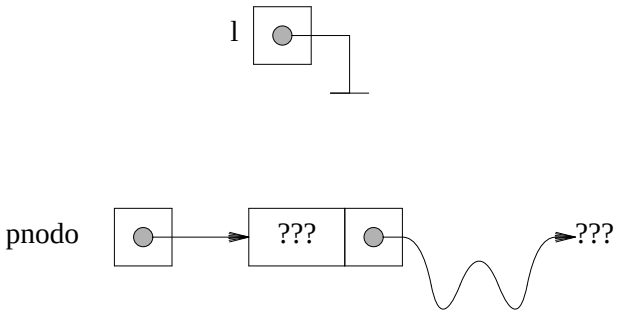
Si llamamos a este procedimiento para insertar un `3` en una lista vacía, tendríamos este estado justo en el `begin` del procedimiento:





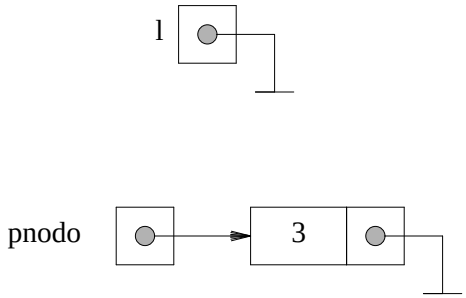
Tras la primera sentencia del procedimiento tenemos:

```
4       pnodo := new TipoNodoEntero;
```



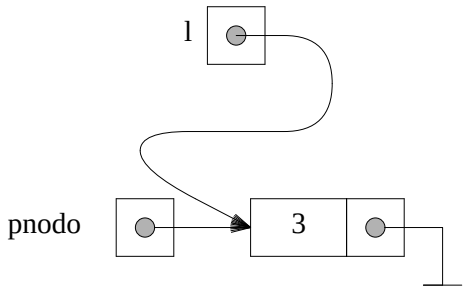
Nótese que no hemos inicializado el nodo. Por tanto, tanto el valor como el puntero al siguiente nodo (contenidos en el nodo) tienen valores aleatorios. Si ejecutamos ahora la segunda y la tercera sentencia del procedimiento tenemos:

```
5       pnodo.valor := n;
6       pnodo.siguiente := l;
```



El nodo está ahora bien inicializado. Lo que falta es cambiar la lista para que apunte al nuevo nodo, cosa que hace la última sentencia del procedimiento:

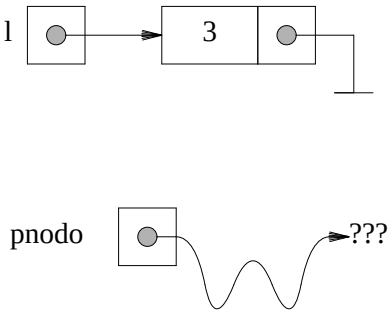
```
7       l := pnodo;
```



Ahora retornamos de *InsertarEntero* y la lista *l* se queda perfectamente definida con un nuevo elemento.

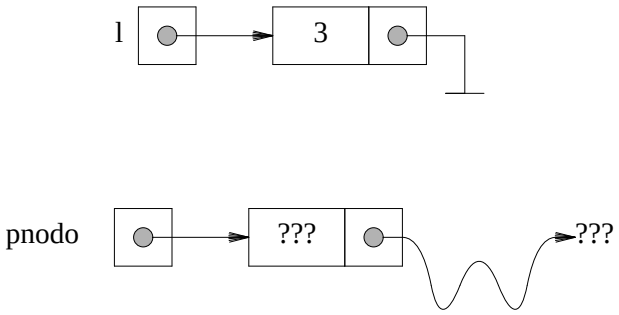
Nótese que, al programar con listas, es preciso distinguir entre el caso en que la lista se encuentra vacía y el caso en que no lo está. En el procemiento anterior el código funciona en ambos casos, pero en general es preciso comprobar si la lista está vacía o no y hacer una cosa u otra según el caso.

Veamos qué sucede al insertar en número 4 en la lista enlazada según ha quedado tras la inserción anterior. Inicialmente, en el *begin* del procedimiento, tenemos:



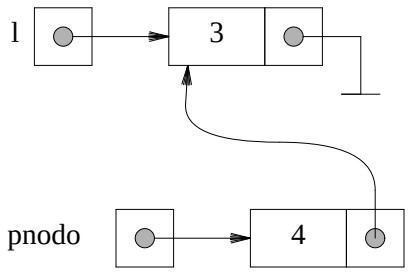
Al ejecutar...

```
4       pnodo := new TipoNodoEntero;
```



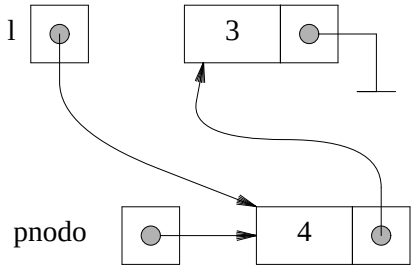
Al ejecutar...

```
5       pnodo.valor := n;
6       pnodo.siguiente := l;
```

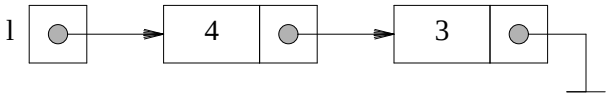


Y por último...

```
7       l := pnodo;
```



Cuando el procedimiento *InsertarEntero* termine la variable *pnodo* termina su existencia y la lista es en realidad:



Un detalle importante que el lector astuto habrá notado es que, si a un procedimiento se le pasa como parámetro por valor una lista, dicho procedimiento puede siempre alterar el contenido de la lista. La lista se podrá pasar por valor o por referencia, pero los elementos que están en la lista siempre se pasan por referencia. Al fin y al cabo una lista es una referencia a un nodo.

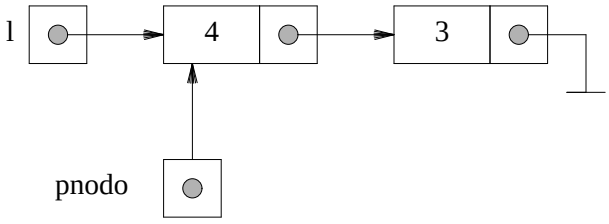
El procedimiento para eliminar un nodo de la cabeza de la lista podría ser como sigue:

```
1     procedure EliminarCabeza(l: in out TipoListaEnteros) is
2          pnodo: TipoListaEnteros;
3     begin
4          if not EsListaEnterosVacia(l) then
5               pnodo := l;
6               l := l.siguiente;
7               Delete(pnodo);
8          end if;
9     end;
```

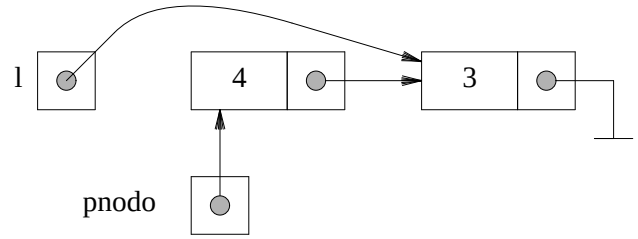
Aquí hay varios detalles importantes. Por un lado, hay que comprobar si la lista está vacía o no. No podemos atravesar el puntero de la lista si esta está vacía. Aunque se supone que nadie debería llamar a *EliminarCabeza* en una lista vacía, es mejor comprobarlo y asegurarse.

Otro detalle es que guardamos en *pnodo* el puntero al nodo que estamos eliminando. Una vez hemos avanzado *l* para que apunte al siguiente nodo (o sea *null* si sólo había un nodo), tenemos en *pnodo* el puntero cuyo elemento apuntado hay que liberar, si no queremos perder la memoria del nodo. Por ejemplo, al eliminar el primer nodo de la última lista que teníamos se producen los siguientes estados:

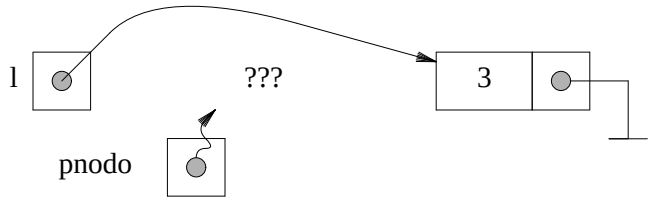
```
5          pnodo := l;
```



```
6       l := l.siguiente;
```



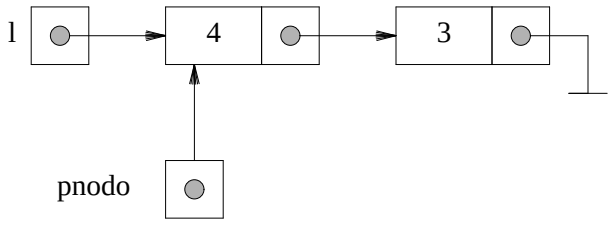

```
7       Delete(pnodo);
```

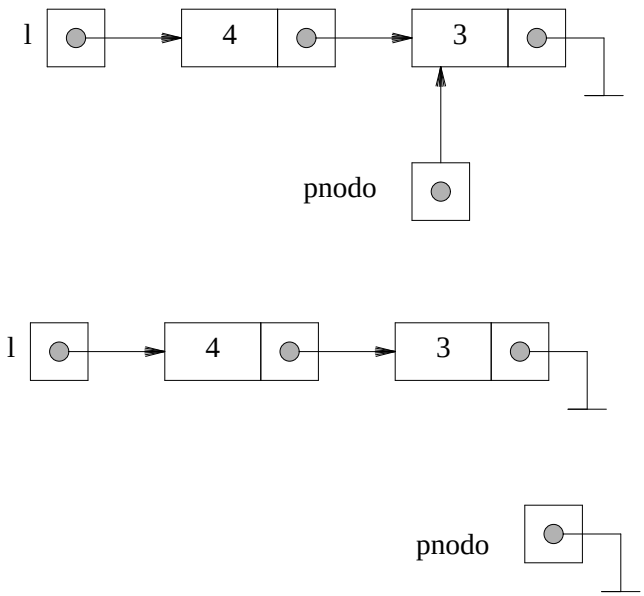


Para continuar con los ejemplos de uso de punteros, el siguiente procedimiento imprime todos los enteros presentes en nuestra lista enlazada.

```
1     procedure EscribirLista(l: TipoListaEnteros) is
2           pnodo: TipoListaEnteros;
3     begin
4           pnodo := l;
5           while not EsListaEnterosVacia(pnodo) loop
6                 Put(pnodo.valor);
7                 pnodo := pnodo.siguiente;
8           end loop;
9     end;
```

Utilizamos *pnodo* para ir apuntando sucesivamente a todos los nodos de la lista. En distintas ejecuciones del *while*, vamos teniendo los siguientes estados (y en cada uno imprimimos el valor guardado en el nodo correspondiente).





Si queremos utilizar la lista enlazada como si fuese una cola (esto es, una estructura de datos *FIFO* o *First In is First Out*, osea, el primero en entrar es el primero en salir) podríamos insertar elementos por el principio como antes y extraerlos por el final. Para hacerlo podríamos implementar un procedimiento *ExtraerCola* que extraiga un elemento de la cola de la lista y devuelva su valor. Este procedimiento puede ser como sigue:

```
1     -- extrae el ultimo elemento.
2     -- no se le puede llamar con una lista vacia.
3     procedure ExtraerCola(l: in out TipoListaEnteros; n: out Integer) is
4          pnodo: TipoListaEnteros;
5          panterior: TipoListaEnteros;
6          ultimo: Bolean;
7     begin
8          if not EsListaEnterosVacia(l) then
9               pnodo := l;
10              if EsListaEnterosVacia(l.siguiente) then
11                   n := l.valor;
12                   l := null;
13              else
14                   panterior := null;
15                   ultimo := False;
16                   loop
17                        ultimo := pnodo.next = null;
18                        if not ultimo then
19                             panterior := pnodo;
20                             pnodo := pnodo.next;
21                        end if;
22                        exit when ultimo;
23                   end loop;
24                   panterior.siguiente := null;
25                   n := pnodo.valor;
26              end if;
27              Delete(pnodo);
28         end if;
29    end;
```

Hay tres casos como puede verse. O la lista está vacía, o tiene un sólo elemento o tiene más de

uno. Hay que distinguir el segundo caso del tercero puesto que en el segundo hay que dejar la lista a *null* y en el tercero hay que dejar a *null* el puntero del último nodo, por lo que no es posible implementar ambos casos del mismo modo si seguimos este algoritmo en el código. Puede verse que en el último caso mantenemos un puntero *panterior* apuntando siempre al nodo anterior a aquel que estamos considerando en el bucle. Lo necesitamos para poder acceder al campo *siguiente* del nodo anterior, que es el que hay que poner a *null*.

Hay formas más compactas de implementar esto pero podrían resultar más complicadas.

11.8. Punteros a variables existentes

Dada cualquier variable ya declarada es posible hacer que un puntero apunte hacia dicha variable. Para ello dicha variable debe declararse empleando la palabra reservada *aliased* y hay que utilizar el atributo *Unchecked_Access* de la variable para obtener su dirección. Por ejemplo:

```
entero: aliased Integer;
pentero: access all Integer;
 ...
pentero := entero'Unchecked_Access;
```

11.9. Invertir la entrada con una pila

Queremos ver si la entrada estándar es un palíndrome, pero sin restricciones respecto a cómo de grande puede ser el texto en la entrada. Como ya sabemos de problemas anteriores un palíndrome es un texto que se lee igual al derecho que al revés.

Una forma de comprobar si un texto es palíndrome es dar la vuelta al texto y compararlo con el original. Ya vimos en un ejercicio anterior que tal cosa puede hacerse empleando una pila (recordamos que una pila es una estructura en la que puede ponerse algo sobre lo que ya hay y extraer lo que hay sobre la pila; igual que en una pila de papeles puede dejarse uno encima o tomar el que hay encima). La figura 11.5 muestra el proceso.

La idea es que podemos ir dejando sobre la pila (insertar en la pila) todos los caracteres del texto, uno por uno. Si los extraemos de la pila nos los vamos a encontrar en orden inverso. De ahí que a las pilas se las llame LIFO, como ya sabemos.

Podemos pues implementar nuestro programa utilizando una pila. Eso sí, puesto que no se permiten restricciones de tamaño (salvo las impuestas por los límites físicos del ordenador) no podemos utilizar estructuras de datos estáticas (por ejemplo un *array*) para implementar la pila. Vamos a utilizar una lista para implementar la pila.

Una lista de caracteres estaría definida de como un puntero a un nodo, el cual tiene un carácter y un puntero al resto de la lista.

```
1     type TipoNodoCar;

3     type TipoListaCar is access all TipoNodoCar;

5     type TipoNodoCar is
6     record
7          c: Character;
8          siguiente: TipoListaCar;
9     end record;
```

Pero para que quede claro que estamos manipulando una pila de caracteres vamos a definir un nuevo nombre de (sub)tipo.

```
11    subtype TipoPilaCar is TipoListaCar;
```

Tradicionalmente se denomina *push* a la operación de insertar algo en una pila y *pop* a la

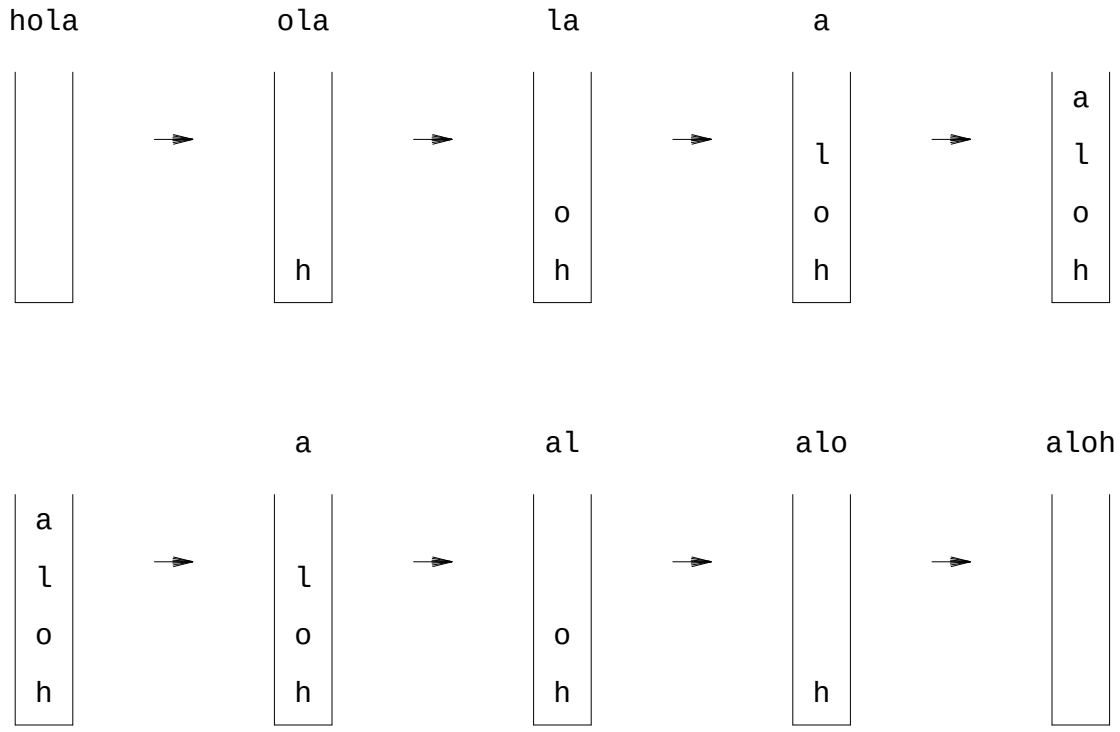


Figura 11.5: *Invertir un texto usando una pila.*

operación de extraer algo de una pila, por lo que vamos a utilizar estos nombres. Teniendo en cuenta esto, nuestro programa principal sería algo así:

```
1          pila: TipoPilaCar;
2          c: Character;
3     begin
4          pila := NuevaPila;
5          while not End_Of_File loop
6               Get(c);
7               Push(pila, c);
8          end loop;
9          while not EsVacia(pila) loop
10              Pop(pila, c);
11              Put(c);
12         end loop;
13    end;
```

Como mostraba la figura, introducimos todo el texto en una pila (incluyendo saltos de línea) y luego lo extraemos e imprimimos.

Para implementar *Push* tenemos que tomar la pila y pedir memoria dinámica para un nodo. Dicho nodo hay que insertarlo al principio de la lista enlazada utilizada para implementar la pila.

```
1     procedure Push(pila: in out TipoPilaCar; c: Character) is
2          pnodo: TipoListaCar;
3     begin
4          pnodo := new TipoNodoCar;
5          pnodo.c := c;
6          pnodo.siguiente := pila;
7          pila := pnodo;
8     end;
```

Por cierto, recuerda que expresiones como *pnodo.c* son en realidad *pnodo.all.c* y están atravesando un puntero. Este ejemplo se refiere al campo *c* del *record* al que apunta el puntero *pnodo.*

Para implementar *Pop* tenemos que guardar en un puntero auxiliar el primer nodo de la lista. Una vez lo hemos extraído de la lista podemos liberarlo.

```
1     procedure Pop(pila: in out TipoPilaCar; c: out Character) is
2          pnodo: TipoListaCar;
3     begin
4          c := pila.c;
5          pnodo := pila;
6          pila := pila.siguiente;
7          Delete(pnodo);
8     end;
```

Una mejora que deberíamos hacer es manejar con cuidado los saltos de línea. Podemos ver si en la entrada encontramos uno y en tal caso insertar en la pila un carácter como por ejemplo *ASCII.LF*. Si al extraer caracteres encontramos dicho carácter entonces reproducimos el salto de línea. Ahora el programa funciona como esperamos:

```
; ada invertir.adb
; invertir
hola
que tal
Control-d
lat euq
aloh
```

Y este es el programa:

invertir.adb

```
1     --
2     -- Invertir la entrada estandar
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     with Ada.Unchecked_Deallocation;

9     procedure Invertir is

11         type TipoNodoCar;

13         type TipoListaCar is access all TipoNodoCar;

15         type TipoNodoCar is
16         record
17              c: Character;
18              siguiente: TipoListaCar;
19         end record;
```

```
21          subtype TipoPilaCar is TipoListaCar;

23          procedure Delete is new Ada.Unchecked_Deallocation(TipoNodoCar, TipoListaCar);
24          function NuevaPila return TipoPilaCar is
25          begin
26               return null;
27          end;

29          function EsVacia(pila: TipoPilaCar) return Boolean is
30          begin
31               return pila = null;
32          end;

34          procedure Push(pila: in out TipoPilaCar; c: Character) is
35               pnodo: TipoListaCar;
36          begin
37               pnodo := new TipoNodoCar;
38               pnodo.c := c;
39               pnodo.siguiente := pila;
40               pila := pnodo;
41          end;

43          procedure Pop(pila: in out TipoPilaCar; c: out Character) is
44               pnodo: TipoListaCar;
45          begin
46               c := pila.c;
47               pnodo := pila;
48               pila := pila.siguiente;
49               Delete(pnodo);
50          end;

52          pila: TipoPilaCar;
53          c: Character;
54     begin
55          pila := NuevaPila;
56          while not End_Of_File loop
57               if End_Of_Line then
58                    Skip_Line;
59                    c := ASCII.LF;
60               else
61                    Get(c);
62               end if;
63               Push(pila, c);
64          end loop;
65          while not EsVacia(pila) loop
66               Pop(pila, c);
67               if c = ASCII.LF then
68                    New_Line;
69               else
70                    Put(c);
71               end if;
72          end loop;
73     end;
```

11.10. ¿Es la entrada palíndrome?

Para ver si la entrada (sea lo extensa que sea) es palíndrome podemos invertirla y compararla con la entrada original. Podemos hacer esto utilizando además de una pila otra estructura de datos, llamada cola, que mantiene el orden de entrada. Una cola es similar a las que puedes ver en los bancos. Los elementos (la gente en los bancos) llegan en un orden determinado a la cola y salen en el orden en que han entrado.

El plan es insertar a la vez cada carácter que leamos de la entrada tanto en una pila como en una cola.

```
1     pila := NuevaPila;
2     cola := NuevaCola;
3     while not End_Of_File loop
4          if End_Of_Line then
5               Skip_Line;
6               c := ASCII.LF;
7          else
8               Get(c);
9          end if;
10         Push(pila, c);
11         Insertar(cola, c);
12    end loop;
```

Luego extraemos un carácter de la pila y otro de la cola y los comparamos, lo que compara el primer carácter con el último (dado que la pila invierte el orden y la cola lo mantiene). Hacemos esto con todos los caracteres. Si en algún caso los caracteres no coinciden es que la entrada no es palíndrome.

```
13    espalindrome := True;
14    while not EsPilaVacia(pila) and not EsColaVacia(cola) and espalindrome loop
15         Pop(pila, c);
16         Extraer(cola, c2);
17         espalindrome := c = c2;
18    end loop;
```

Por supuesto, al final, tanto la pila como la cola deberán estar vacías, dado que hemos insertado el mismo número de caracteres en ambas.

Para implementar la cola podemos insertar elementos por el final (como en los bancos) y extraerlos por el principio (idem). Para esto resulta útil mantener un puntero al último elemento además del habitual puntero al primer elemento. La figura 11.6 muestra nuestra implementación para la cola.

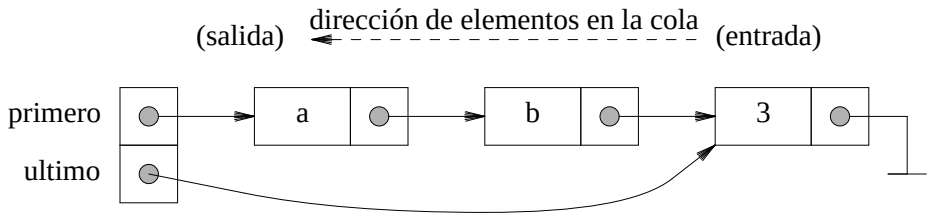


Figura 11.6: *Nuestra implementación para la cola con un puntero al primer nodo y otro al último.*

Cuando un nuevo elemento llegue a la cola lo vamos a insertar justo tras *ultimo*. Cuando queramos sacar un elemento de la cola lo vamos a extraer justo por el *primero*. Por lo tanto, el tipo de datos para la cola utiliza el tipo de datos para la lista de caracteres que hemos definido antes, pero corresponde ahora a un registro con punteros al primero y al último.

```
1     type TipoColaCar is
2     record
3          primero: TipoListaCar;
4          ultimo: TipoListaCar;
5     end record;
```

Extraer un elemento de la cola es exactamente igual a hacer un *Pop* de una pila. Salvo por que si la cola se queda vacía tenemos que dejar ambos punteros a *null* (el primero y el último).

Insertar un elemento en la cola consiste en enlazarlo justo detrás del nodo al que apunta *ultimo*. No obstante, si la cola está vacía hay que poner ambos punteros (el primero y el último) apuntando al nuevo nodo.

El programa queda como se muestra a continuación.

invertir.adb

```
1     --
2     -- Invertir la entrada estandar
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     with Ada.Unchecked_Deallocation;

9     procedure Invertir is

11         type TipoNodoCar;

13         type TipoListaCar is access all TipoNodoCar;

15         type TipoNodoCar is
16         record
17              c: Character;
18              siguiente: TipoListaCar;
19         end record;

21         subtype TipoPilaCar is TipoListaCar;

23         type TipoColaCar is
24         record
25              primero: TipoListaCar;
26              ultimo: TipoListaCar;
27         end record;

29         procedure Delete is new Ada.Unchecked_Deallocation(TipoNodoCar, TipoListaCar);
30         function NuevaPila return TipoPilaCar is
31         begin
32              return null;
33         end;

35         function NuevaCola return TipoColaCar is
36              cola: TipoColaCar;
37         begin
38              cola.primero := null;
39              cola.ultimo := null;
40              return cola;
41         end;
```

```
43          function EsPilaVacia(pila: TipoPilaCar) return Boolean is
44          begin
45               return pila = null;
46          end;

48          function EsColaVacia(cola: TipoColaCar) return Boolean is
49          begin
50               return cola.primero = null;
51          end;

53          procedure Push(pila: in out TipoPilaCar; c: Character) is
54               pnodo: TipoListaCar;
55          begin
56               pnodo := new TipoNodoCar;
57               pnodo.c := c;
58               pnodo.siguiente := pila;
59               pila := pnodo;
60          end;

62          procedure Insertar(cola: in out TipoColaCar; c: Character) is
63               pnodo: TipoListaCar;
64          begin
65               pnodo := new TipoNodoCar;
66               pnodo.c := c;
67               pnodo.siguiente := null;
68               if EsColaVacia(cola) then
69                    cola.primero := pnodo;
70               else
71                    cola.ultimo.siguiente := pnodo;
72               end if;
73               cola.ultimo := pnodo;
74          end;

76          procedure Pop(pila: in out TipoPilaCar; c: out Character) is
77               pnodo: TipoListaCar;
78          begin
79               c := pila.c;
80               pnodo := pila;
81               pila := pila.siguiente;
82               Delete(pnodo);
83          end;

85          procedure Extraer(cola: in out TipoColaCar; c: out Character) is
86               pnodo: TipoListaCar;
87          begin
88               c := cola.primero.c;
89               pnodo := cola.primero;
90               if cola.ultimo = cola.primero then
91                    cola.ultimo := null;
92               end if;
93               cola.primero := cola.primero.siguiente;
94               Delete(pnodo);
95          end;
```

```
97          pila: TipoPilaCar;
98          cola: TipoColaCar;
99          c: Character;
100         c2: Character;
101         espalindrome: Boolean;

103    begin
104         pila := NuevaPila;
105         cola := NuevaCola;
106         while not End_Of_File loop
107              if End_Of_Line then
108                   Skip_Line;
109                   c := ASCII.LF;
110              else
111                   Get(c);
112              end if;
113              Push(pila, c);
114              Insertar(cola, c);
115         end loop;

117         espalindrome := True;
118         while not EsPilaVacia(pila) and not EsColaVacia(cola) and espalindrome loop
119              Pop(pila, c);
120              Extraer(cola, c2);
121              espalindrome := c = c2;
122         end loop;

124         if espalindrome then
125              Put("palindrome");
126         else
127              Put("no palindrome");
128         end if;
129         New_Line;
130    end;
—
```

Problemas

Como de costumbre, algunos enunciados corresponden a problemas hechos con anterioridad. Sugerimos que los vuelvas a hacer sin mirar en absoluto las soluciones. Esta vez sería deseable que compares luego tus soluciones con las mostradas en el texto.

1 Imprimir los números de la entrada estándar al revés, sin límite en el número de números que se pueden leer.

2 Leer una palabra de la entrada estándar y escribirla en mayúsculas, sin límite en la longitud de dicha palabra.

3 Implementar un conjunto de enteros con operaciones necesarias para manipularlo y sin límite en el número de enteros que puede contener.

4 Imprimir un histograma que indique cuántas veces se repite cada palabra en la entrada estándar, sin límite en el número de palabras que puede haber.

5 Imprimir un histograma que indique cuántas veces se repite cada palabra en la entrada estándar, sin límite en el número de palabras que puede haber y sin límite en cuanto a la longitud de la palabra.

6 Implementar un pila utilizando una lista enlazada. Utilizarla para ver si la entrada estándar es palíndroma.

7 Implementar operaciones aritméticas simples (suma y resta) para números de longitud

indeterminada.

8 Modifica la calculadora realizada anteriormente para que incluya un historial de resultados. Deben incluirse dos nuevos comandos: *pred* y *succ*. El primero debe tener como valor el resultado anterior al último mostrado. El segundo debe tener como valor el resultado siguiente al último mostrado. Deben permitirse historias arbitrariamente largas.

9 Modificar el juego del bingo para que permita manipular cualquier número de cartones.

10 Implementar un tipo de datos y operaciones para manipular *strings* de longitud variable de tal forma que cada *string* esté representado por una lista de *arrays*. La idea es que cada nodo en la lista contiene un número razonable de caracteres (por ejemplo 50) de tal forma que para la lista tiene pocos nodos pero aun así puede crecer y además no es preciso copiar todo el contenido si se quiere hacer crecer una palabra.

11 Un analizador léxico es un programa que lee de la entrada una secuencia de caracteres y produce como resultado una secuencia de lexemas o tokens. Escribir un analizador léxico que distinga a la entrada, tomada del fichero ‘‘`datos.txt`’’, los siguientes tokens: Identificadores (palabras que comienzan por una letra mayúscula o minúscula y están formadas por letras, números y subrayados); Números (Uno o más dígitos, opcionalmente seguidos de un ‘‘`.`’’ y uno o más dígitos). Operadores (uno de ‘‘`+`’’, ‘‘`-`’’, ‘‘`*`’’, ‘‘`/`’’); La palabra reservada ‘‘`begin`’’; La palabra reservada ‘‘`end`’’; Comentarios (dos ‘‘`-`’’ y cualquier otro carácter hasta el fin de línea).

El analizador debe construir una lista enlazada de tokens donde cada token debe al menos indicar su tipo y posiblemente un valor real o de cadena de caracteres para el mismo, en aquellos casos en que un token pueda corresponder a distintos valores. Además de un procedimiento que construya la lista de tokens es preciso implementar otro que cada vez que se le invoque devuelva el siguiente token sin necesidad de recorrer la lista desde el principio cada vez. Se sugiere mantener una estructura de datos para recordar la última posición devuelta por este subprograma.

12 — E es el editor definitivo

12.1. Un editor de línea

Hace mucho tiempo se utilizaban editores de línea, llamados así puesto que sólo eran capaces de ejecutar en la llamada línea de comandos (lo que Windows llama ‘‘símbolo del sistema’’ y MacOS X llama ‘‘terminal’’). Se trata de editores capaces de utilizar la entrada estándar para aceptar órdenes y texto y la salida estándar para mostrar texto. No utilizan ventanas ni paneles gráficos de texto ni operan a pantalla completa.

Queremos implementar un editor de línea, que llamaremos *e*, capaz de editar múltiples ficheros. El editor debe mantener en la memoria el contenido de los ficheros que esté editando, utilizando la estructura de datos que resulte oportuna.

Cuando se le invoque, el editor debe leer órdenes de la entrada estándar y actuar en consecuencia. Deseamos tener las siguientes órdenes en el editor:

`e fichero.txt`
: Comienza a editar el fichero `fichero.txt` (o cualquier otro cuyo nombre se indique). A partir de este momento los comandos de edición se refieren a dicho fichero.

`w`
: Actualiza el fichero que se está editando, escribiendo el contenido.

`f`
: Escribe el nombre del fichero que se está editando.

`x`
: Escribe el nombre de todos los ficheros que se están editando.

`d numero, numero`
: Borra las líneas comprendidas entre los dos números de línea (inclusive ambos).

`d numero`
: Borra la línea cuyo número se indica.

`i numero`
: Lee líneas y las inserta antes de la línea cuyo número se indica. Se supone que las líneas leídas terminan en una única línea cuyo contenido es el carácter ‘‘`.`’’.

`i numero, numero`
: Reemplaza las líneas comprendidas entre los dos números (inclusive ambos) por las líneas leídas de la entrada (como el comando anterior).

`p`
: Imprime el contenido del fichero entero en la salida.

`p numero`
: Imprime el contenido de la línea cuyo número se indica.

`p numero, numero`
: Imprime el contenido de las líneas comprendidas entre los dos números (inclusive).

`s/palabra/otra/`
: Reemplaza ‘‘`palabra`’’ por ‘‘`otra`’’ en el fichero. Donde ‘‘`palabra`’’ debe de ser una palabra en el texto. Por ejemplo, si ‘‘`palabra`’’ forma parte de otra palabra más larga entonces esa otra palabra debe dejarse inalterada.

Ni que decir tiene que no queremos que el editor tenga límites respecto al tamaño de los ficheros que edita, de las líneas de estos ficheros o de las palabras que lo componen.

12.2. ¿Por dónde empezamos?

El problema parece complicado y, desde luego, lo primero es simplificarlo en extremo hasta que tengamos algo mas manejable.

La primera simplificación es pensar en un único fichero, incluso asumiendo que siempre va a ser, digamos, un fichero llamado `datos.txt` el que vamos a editar.

Por lo demás podríamos olvidarnos de todos los comandos que hay que implementar y empezar por definir una estructura de datos capaz de mantener el fichero en la memoria e imprimirlo, pero pensando en que vamos a querer utilizar dicha estructura para hacer el tipo de cosas que indican los comandos.

A la vista de los comandos parece que vamos a tener que manipular palabras y líneas. Podríamos suponer que va a resultar adecuado implementar un fichero como una serie de líneas y una línea como una serie de palabras. Además, dado que no podemos tener límites en cuanto a tamaño, tanto el tipo de datos para una palabra, como el de una línea y el del fichero han de ser estructuras de datos dinámicas, capaces de crecer en tiempo de ejecución.

Visto todo esto podríamos empezar por definir e implementar un tipo de datos para palabras de tamaño dinámico. Luego podríamos utilizar esto para construir líneas y ficheros. Por último podemos preocuparnos de ir implementando todos los comandos que necesitemos.

12.3. Palabras de tamaño variable

La forma más simple de implementar un *string* dinámico, o de tamaño variable, es utilizar un puntero que apunte a un *string* estático. Según nuestras necesidades podemos pedir un *string* del tamaño adecuado y utilizar siempre el puntero a dicho *string* para manipularlo. La figura 12.1 muestra varios *strings* dinámicos.

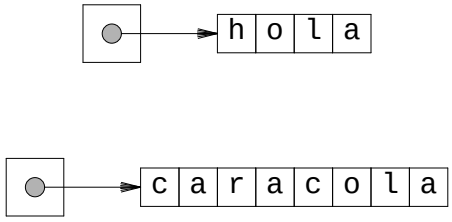


Figura 12.1: *Varios strings dinámicos*

En la figura, el *string* dinámico de arriba apunta a un *string* (estático) de cuatro caracteres. El de abajo en cambio apunta a un *string* de ocho caracteres. No obstante, los dos objetos son lo mismo: básicamente un puntero al almacenamiento que se utiliza para mantener los caracteres.

El problema es que conforme leamos palabras y manipulemos nuestros *strings* vamos a querer hacerlos crecer y decrecer. En lugar de estar todo el tiempo creando *strings* estáticos justo del tamaño adecuado podemos hacer que cada *string* dinámico contenga (además del puntero a los caracteres) el tamaño disponible para almacenar caracteres y el tamaño realmente utilizado. Esto es similar a lo que hicimos para almacenar palabras utilizando un *array* o *string* estático. La diferencia es que en este caso el tamaño de nuestro almacén puede variar.

Podríamos pues comenzar por declarar:

```
1    type TipoPtrString is access all String;
```

```
3     type TipoString is
4     record
5          cars: TipoPtrString;
6          long: Natural;
7          max: Natural;
8     end record;
```

Aquí, *cars* es el puntero al almacén de caracteres, *long* es la longitud del *string* y *max* es el tamaño del almacén (el número máximo de caracteres que podemos almacenar con la memoria a la que apunta *cars*). Hemos declarado el puntero como un puntero a *String* para poder hacer que apunte a *strings* de distinto tamaño (igual que hacemos con los parámetros de las funciones). La figura 12.2 muestra los mismos *strings* de la figura 12.1 pero utilizando nuestro tipo de datos.

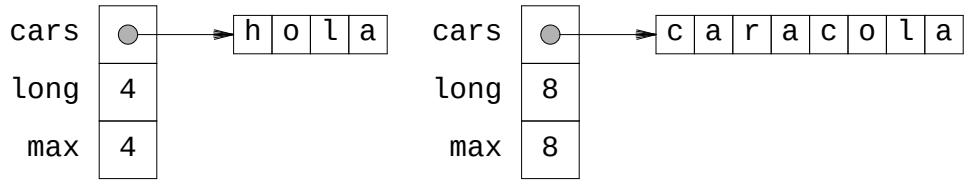


Figura 12.2: *Versión refinada de los mismos strings.*

Si ahora copiamos, por ejemplo, el string de la izquierda al de la derecha el resultado podría ser el que muestra la figura 12.3.

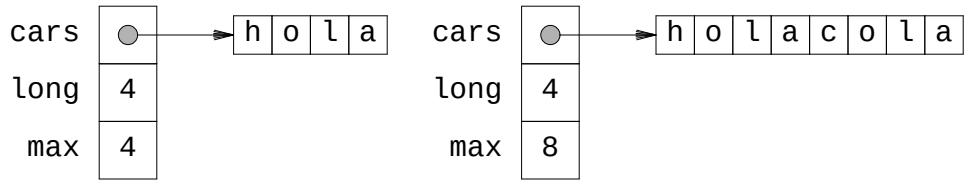


Figura 12.3: *Resultado tras copiar el string de la izquierda al de la derecha.*

El *string* de la derecha sigue utilizando el mismo almacén, pero su longitud oficial es ahora 4 y, además, los primeros cuatro caracteres son los que corresponden al *string* de la izquierda. Esta es la idea. Mientras no necesitemos más capacidad en el almacén utilizaremos el que ya tiene el *string*.

Lo que estamos intentando hacer es implementar un tipo *TipoString* que podamos utilizar casi del mismo modo que podemos utilizar los que tenemos en Ada, salvo por que este será capaz de cambiar de tamaño. Por el momento es bueno olvidarse del resto del problema e intentar implementar una buena abstracción para manipular palabras.

Para crear un nuevo *string* dinámico podemos implementar una función que devuelva un *TipoString*. Pero en lugar de hacer que devuelva uno vacío (como de costumbre) vamos a hacer que devuelva uno que contenga inicialmente lo que le digamos.

```
1     function NuevoString(ini: String)  return TipoString is
2          s: TipoString;
3     begin
4          s.cars := new String(1..ini'Length);
5          s.long := ini'Length;
6          s.max := ini'Length;
7          s.cars(1..s.long) := ini;
8          return s;
9     end;
```

La línea 4 de esta función pide memoria para un *array* de caracteres de la longitud requerida para

almacenar una copia del argumento. Por tanto, inicializamos tanto *long* como *max* a dicha longitud, copiamos los caracteres y devolvemos el nuevo *string* dinámico. No ha sido difícil.

Como no queremos tener que mirar dentro de variables de tipo *TipoString* vamos a definir también funciones auxiliares. Esta devuelve qué longitud tiene un *string*:

```
1     function LongString(s: TipoString) return Natural is
2     begin
3          return s.long;
4     end;
```

Esta compara un *string* con otro:

```
1     function IgualString(s1: TipoString; s2: TipoString) return Boolean is
2     begin
3          if s1.long /= s2.long then
4               return False;
5          else
6               return s1.cars(1..s1.long) = s2.cars(1..s2.long);
7          end if;
8     end;
```

Hay que tener cuidado de asegurarse de que los dos son de la misma longitud antes de compararlos. Piensa que ahora nuestros *strings* son de cualquier tamaño. Y además, sólo deben compararse los primeros *long* caracteres (puesto que el almacén puede contener más caracteres).

Copiar un *string* en otro es un poco delicado. Para empezar, no podemos utilizar la asignación de Ada para copiar un *TipoString* en otro. De hacerlo estaríamos copiando el puntero al almacén pero no el almacén. Además, estaríamos perdiendo el antiguo almacén del destino. Vamos, un desastre. Siempre que tenemos estructuras dinámicas (con punteros) debemos tener cuidado de suministrar operaciones para copiarlas.

Al copiar el contenido de un *string* en otro hay que tener cuidado. La cuestión es ver si tenemos suficiente espacio en el almacén del destino (como en el caso de la figura 12.3). Si lo tenemos basta con copiar los caracteres y ajustar la longitud. En otro caso hay que pedir un nuevo almacén del tamaño apropiado y recordar que antes debemos deshacernos del antiguo.

```
1     procedure CopiarString(dest: in out TipoString; orig: TipoString) is
2     begin
3          if dest.max < orig.long then
4               Delete(dest.cars);
5               dest.cars := new String(1..orig.long);
6               dest.max := orig.long;
7          end if;
8          dest.cars(1..orig.long) := orig.cars.all;
9          dest.long := orig.long;
10    end;
```

Puede resultar igualmente útil añadir un *string* a la derecha de otro. A eso se le llama hacer un *append* (añadir) de un string. Para hacer esto tenemos casi el mismo problema que en el caso de copiar un *string* a otro. La diferencia radica en que ahora tenemos que conservar los caracteres que había en el *string*. (Hay que añadir más, pero no borrar ninguno). Si falta memoria en el almacén y hay que pedir más hay que recordar que tenemos que copiar los caracteres antiguos al nuevo almacén, antes de añadir los nuevos.

```
1     procedure AppString(dest: in out TipoString; orig: TipoString) is
2          paux: TipoPtrString;
3          nuevolong: Natural;
4     begin
5          nuevolong := dest.long + orig.long;
6          if dest.max < nuevolong then
7               paux := dest.cars;
8               dest.cars := new String(1..nuevolong);
9               dest.max := nuevolong;
10              dest.cars(1..dest.long) := paux(1..dest.long);
11              Delete(paux);
12         end if;
13         dest.cars(dest.long+1..nuevolong) := orig.cars.all;
14         dest.long := nuevolong;
15    end;
```

Tampoco debemos olvidarnos de que, dado que nuestro *string* es dinámico, tendremos que destruirlo cuando no sea de utilidad. Para eso podemos implementar un procedimiento *DestruirString* que libere el almacén. Recuerda que no queremos estar pensando en el resto del programa cómo están hechos estos *strings*. Por eso damos esta operación en lugar de dejar que otras partes del programa liberen ellos mismos el almacén.

```
1     procedure DestruirString(s: in out TipoString) is
2     begin
3          if s.cars /= null then
4               Delete(s.cars);
5               s.cars := null;
6               s.max := 0;
7          end if;
8     end;
```

Fíjate en que sólo liberamos el almacén si había uno asignado. Podría ser que el *string* siempre haya estado vacío y nunca haya tenido almacén alguno. Además, dejamos su tamaño y longitud a cero, para que todo sea coherente con un *string* que no tiene espacio. No queremos sorpresas.

¿Y si se nos olvida algo? Un buen truco es obtener un *string* de los de Ada a partir del nuestro. Utilizando dicho *string* podemos hacer con él cosas que el lenguaje sabe hacer pero que no hemos programado.

```
1     function AdaString(s: TipoString) return String is
2     begin
3          if s.long = 0 then
4               return "";
5          else
6               return s.cars(1..s.long);
7          end if;
8     end;
```

¡Cuidado! Si el *string* estaba vacío puede que no tuviese almacén. En tal caso no podemos utilizar *s.cars* y es mejor devolver un *string* vacío (de Ada) directamente. Esta función puede ser muy ineficiente puesto que hace una copia de los caracteres para devolverlos, por eso debería ser el último recurso y por eso hemos implementado el resto de operaciones en lugar de utilizar siempre esta.

Podríamos ahora empezar por leer nuestro fichero `datos.txt` en una sóla palabra e imprimirlo, para ejercitar nuestro nuevo tipo de datos y sus operaciones. Este es el programa resultante.

editor.adb

```
1     --
2     -- Editar la entrada
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;

7     with Ada.Unchecked_Deallocation;

9     procedure Editor is

11          type TipoPtrString is access all String;

13          type TipoString is
14          record
15               cars: TipoPtrString;
16               long: Natural;
17               max: Natural;
18          end record;


21          procedure Delete is new Ada.Unchecked_Deallocation(String, TipoPtrString);

23          function NuevoString(ini: String)  return TipoString is
24               s: TipoString;
25          begin
26               s.cars := new String(1..ini'Length);
27               s.long := ini'Length;
28               s.max := ini'Length;
29               s.cars(1..s.long) := ini;
30               return s;
31          end;

33          function LongString(s: TipoString) return Natural is
34          begin
35               return s.long;
36          end;

38          function AdaString(s: TipoString) return String is
39          begin
40               if s.long = 0 then
41                    return "";
42               else
43                    return s.cars(1..s.long);
44               end if;
45          end;

47          function IgualString(s1: TipoString; s2: TipoString) return Boolean is
48          begin
49               if s1.long /= s2.long then
50                    return False;
51               else
52                    return s1.cars(1..s1.long) = s2.cars(1..s2.long);
53               end if;
54          end;
```

```
56          procedure DestruirString(s: in out TipoString) is
57          begin
58               if s.cars /= null then
59                    Delete(s.cars);
60                    s.cars := null;
61                    s.max := 0;
62               end if;
63          end;

65          procedure CopiarString(dest: in out TipoString; orig: TipoString) is
66          begin
67               if dest.max < orig.long then
68                    Delete(dest.cars);
69                    dest.cars := new String(1..orig.long);
70                    dest.max := orig.long;
71               end if;
72               dest.cars(1..orig.long) := orig.cars.all;
73               dest.long := orig.long;
74          end;

76          procedure AppString(dest: in out TipoString; orig: TipoString) is
77               paux: TipoPtrString;
78               nuevolong: Natural;
79          begin
80               nuevolong := dest.long + orig.long;
81               if dest.max < nuevolong then
82                    paux := dest.cars;
83                    dest.cars := new String(1..nuevolong);
84                    dest.max := nuevolong;
85                    dest.cars(1..dest.long) := paux(1..dest.long);
86                    Delete(paux);
87               end if;
88               dest.cars(dest.long+1..nuevolong) := orig.cars.all;
89               dest.long := nuevolong;
90          end;

92          texto: TipoString;
93          fich: File_Type;
94          nuevotexto: TipoString;
95          nuevoc: String(1..1);
96     begin
97          texto := NuevoString("");
98          Open(fich, In_File, "datos.txt");
99          while not End_Of_File(fich) loop
100              if End_Of_Line(fich) then
101                   nuevoc(1) := ASCII.LF;
102                   Skip_Line(fich);
103              else
104                   Get(fich, nuevoc);
105              end if;
106              nuevotexto := NuevoString(nuevoc);
107              AppString(texto, nuevotexto);
108              DestruirString(nuevotexto);
109         end loop;
110         Put(AdaString(texto));
111         DestruirString(texto);
112    end;
—
```

Como puede verse resulta un tanto incómodo añadir un nuevo carácter a nuestro *string*. Hay que guardar el carácter en un *string* de Ada y luego crear uno de nuestros *strings* para llamar finalmente a *AppString*.

Aunque las operaciones no estaban mal, parece que sería mejor tener una operación que deje añadir un carácter a un *string*. Y eso es lo que vamos a hacer.

Ahora tanto la operación de añadir un carácter como la de añadir un *string* deben de hacer crecer el almacén si es necesario. Por lo tanto hemos factorizado el código necesario para hacerlo crecer creando una nueva operación auxiliar, llamada *CrecerString*, que hace crecer el almacén tantos caracteres como indica su parámetro. También hemos cambiado *AppString* para que utilice esta nueva operación. Así quedaría ahora *AppString*:

```
1     procedure AppString(dest: in out TipoString; orig: TipoString) is
2          nuevolong: Natural;
3     begin
4          nuevolong := dest.long + orig.long;
5          if dest.max < nuevolong then
6               CrecerString(dest, orig.long);
7          end if;
8          dest.cars(dest.long+1..nuevolong) := orig.cars.all;
9          dest.long := nuevolong;
10    end;
```

La nueva operación es similar:

```
1     procedure AppChar(dest: in out TipoString; c: Character) is
2     begin
3          if dest.max < dest.long + 1 then
4               CrecerString(dest, Incr);
5          end if;
6          dest.long := dest.long + 1;
7          dest.cars(dest.long) := c;
8     end;
```

Para que no tentamos que copiar todos los caracteres del *string* cada vez que queremos añadir un carácter vamos a hacer crecer el almacén de *Incr* en *Incr* caracteres (utilizando una constante para definir cuánto es este incremento). Así, sólo una de cada *Incr* veces necesitaremos realmente hacer crecer el *string* al añadir un carácter. Hemos definido *Incr* como 32, pero cualquier otro valor podría ser válido.

```
1     -- Los TipoString crecen de Incr en Incr caracteres
2     -- al ir llamando a AppChar
3     Incr: constant Integer := 32;
```

El propósito de esta constante no es obvio en absoluto, por lo que merece un comentario.

Realmente siempre se puede mejorar el conjunto de operaciones. La clave está en partir de un conjunto razonable y en algún momento decir basta y dejar de mejorarlo. Por ejemplo, dado que podemos querer añadir caracteres, *strings* de Ada o nuestros *strings* podríamos en realidad implementar una única operación básica *AppStringAda* y hacer que tanto *AppChar* como *AppString* se limitasen a hacer su trabajo llamando a *AppStringAda*. Nosotros vamos a parar aquí y aceptar el conjunto de operaciones de *TipoString* en su forma actual.

Con nuestros nuevos cambios el programa principal de prueba que teníamos puede ahora quedar mucho más elegante (normalmente se denomina elegantes a los programas sencillos que hacen el trabajo de un modo eficaz).

```
1     texto: TipoString;
2     fich: File_Type;
3     c: Character;
4     begin
5     texto := NuevoString("");
6     Open(fich, In_File, "datos.txt");

8     while not End_Of_File(fich) loop
9          if End_Of_Line(fich) then
10              c := ASCII.LF;
11              Skip_Line(fich);
12         else
13              Get(fich, c);
14         end if;
15         AppChar(texto, c);
16    end loop;

18    Put(AdaString(texto));
19    DestruirString(texto);
20    end;
```

12.4. Líneas y textos

Queríamos nuestros *strings* para poder leer textos y editarlos luego. Es un buen momento para volver a mirar las operaciones de edición que debe aceptar el editor, antes de pensar en qué estructura de datos utilizar para mantener el texto. La operación de sustituir una palabra por otra parece indicar que hemos de ser capaces de manipular por separado las palabras del texto. La mayoría de las operaciones aceptan números de línea, lo que sugiere que hemos de ser capaces de manipular líneas del texto.

Lo mas razonable parece ser utilizar una lista de líneas para representar un texto y una serie de palabras para representar las líneas. Empecemos por definir una línea de texto.

```
1     type TipoNodoPalabra;
2     type TipoListaPalabra is access all TipoNodoPalabra;
3     type TipoNodoPalabra is
4     record
5          palabra: TipoString;
6          siguiente: TipoListaPalabra;
7     end record;

9     type TipoLinea is
10    record
11         primera: TipoListaPalabra;
12         ultima: TipoListaPalabra;
13    end record;
```

Cuando leamos el texto vamos a querer añadir palabras a la línea que estamos leyendo. Por lo tanto estamos utilizando algo muy similar a la cola que implementamos anteriormente. Se trata de una lista enlazada de palabras pero con punteros tanto a la primera como a la última palabra. Así no es preciso recorrer la lista para añadir otra palabra más. La figura 12.4 muestra un ejemplo de dicha estructura.

Y por cierto, ¡Podemos ignorar por completo cómo están hechas las palabras! Ahora una palabra es un *TipoString*. Y no nos importa en absoluto cómo esté hecho. Bueno, sí que nos importa. Pero nos mentimos para poder olvidar todo lo que podamos al respecto. Lo que nos importa ahora son las líneas. Es cierto que sabemos que la estructura que estaremos utilizando para guardar la línea es la mostrada en la figura 12.5, pero jamás pensamos en esos términos. Si nos interesan las palabras pensamos en términos de la figura 12.2 y si nos interesan las líneas

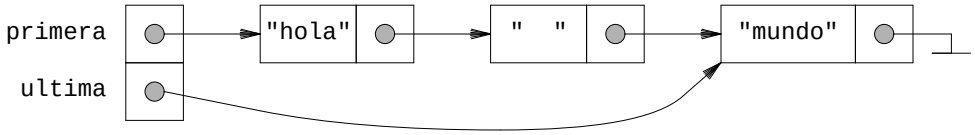


Figura 12.4: *Una línea es una lista de palabras con punteros a la primera y última.*

pensamos en términos de la figura 12.4. Intentamos no tener que pensar siempre en todo a la vez.

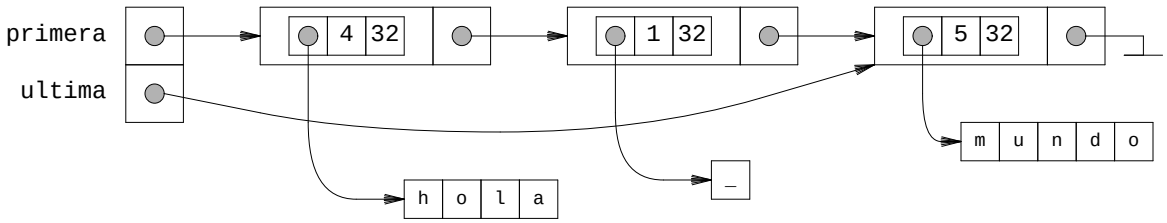


Figura 12.5: *Todos los detalles sobre una línea de texto.*

Por el momento no sabemos mucho respecto a lo que tendremos que hacer con una línea. Podemos realizar operaciones para crear una línea, añadir una palabra a una línea, imprimir una línea y destruir una línea. Si después necesitamos algo más ya nos ocuparemos de ello. Esto crea una línea:

```
1     function NuevaLinea return TipoLinea is
2          nl: TipoLinea;
3     begin
4          nl.primera := null;
5          nl.ultima := null;
6          return nl;
7     end;
```

Añadir una palabra es similar a cuando añadíamos un elemento a una cola. Pero hay un detalle importante. Dado que las palabras son estructuras dinámicas no podemos asignar una palabra a otra. De hacer esto estaríamos copiando los punteros pero no estaríamos copiando el contenido de las palabras. Por eso utilizamos una llamada a *NuevoString* para crear un string con el contenido de la palabra que queremos añadir.

```
1     procedure AppPalabra(linea: in out TipoLinea; palabra: TipoString) is
2          pnodo: TipoListaPalabra;
3     begin
4          pnodo := new TipoNodoPalabra;
5          pnodo.palabra := NuevoString(AdaString(palabra));
6          pnodo.siguiente := null;
7          if linea.primera = null then
8               linea.primera := pnodo;
9          else
10              linea.ultima.siguiente := pnodo;
11         end if;
12         linea.ultima := pnodo;
13    end;
```

Para escribir una línea escribimos una por una todas sus palabras.

```
1     procedure EscribirLinea(linea: TipoLinea) is
2          lista: TipoListaPalabra;
3     begin
4          lista := linea.primera;
5          while lista /= null loop
6               Put(AdaString(lista.palabra));
7               lista := lista.siguiente;
8          end loop;
9          New_Line;
10    end;
```

Destruir la línea requiere recorrerla entera liberando cada palabra y cada nodo de la lista que forma la línea.

```
1     procedure DestruirLinea(linea: in out TipoLinea) is
2          nodo: TipoListaPalabra;
3     begin
4          while linea.primera /= null loop
5               nodo := linea.primera;
6               linea.primera := nodo.siguiente;
7               DestruirString(nodo.palabra);
8               Delete(nodo);
9          end loop;
10         linea.primera := null;
11         linea.ultima := null;
12    end;
```

En este punto tal vez fuese mejor hacer un programa que leyese todo el texto del fichero en una única línea, palabra por palabra.

12.5. Palabras y blancos

Al leer un texto tendremos que distinguir entre palabras y blancos, puesto que el comando ‘‘s’’ del editor exige reemplazar una palabras individuales por otras. Podemos adaptar nuestras funciones para leer palabras y blancos para que ahora lean variables de *TipoString* conteniendo palabras o blancos.

En realidad no hace falta almacenar los blancos puesto que nos bastaría con saber cuántos blancos hay, pero para no complicar aún mas las cosas vamos a utilizar un *string* también para almacenar los blancos.

Este procedimiento salta blancos de la entrada, leyéndolos en un *TipoString*.

```
1     -- Lee un string formado por caracteres distintos a EOL
2     -- pero presentes en sep.
3     procedure LeerBlancos(fich: File_Type; s: out TipoString; sep: String) is
4          fin: Boolean;
5          c: Character;

6     begin
7          s := NuevoString("");
8          fin := False;
```

```
9          while not End_Of_File(fich) and not fin loop
10              Look_Ahead(fich, c, fin);
11              if not fin then
12                   fin := not EstaEn(sep, c);
13              end if;
14              if not fin then
15                   Get(fich, c);
16                   AppChar(s, c);
17              end if;
18         end loop;
19    end;
```

Puesto que ya hemos hecho muchas veces el mismo *LeerBlancos* (en realidad *SaltarBlancos*) esta vez hemos hecho algo distinto. Le suministramos a *LeerBlancos* un *string* que utilizamos como si fuese un conjunto de caracteres. Este *string* contiene los caracteres que se consideran blancos. De este modo podríamos utilizar *LeerBlancos* para leer texto delimitado por los caracteres que queramos considerar blancos. La función *EstaEn* devuelve *True* cuando *c* es un carácter que forma parte de *sep*. Aplazamos su código hasta que luego mostremos el programa completo. No hace falta en realidad saber cómo está hecha para entender código que la use. ¡Eso es abstraer!

Leer una palabra es similar a leer blancos. La única diferencia es que en lugar de detenernos cuando encontremos un carácter que no sea un separador, debemos detenernos cuando el carácter sea un separador. En lugar de implementar otro procedimiento distinto, vamos a cambiar el anterior para que también nos sirva para leer palabras. La idea es que si *invertirsep* es *True* entonces en lugar de considerar *sep* el conjunto de separadores (blancos), consideramos como separadores los caracteres que no están en *sep*. Leer blancos sería leer considerando que todos los caracteres que no son blancos son separadores y nos detienen. Leer una palabra es leer considerando que todos los caracteres que son blancos son separadores y nos detienen. Luego para leer una palabra podríamos llamar a algo como:

```
LeerPalabra(fich, palabra, False, CarsBlancos);
```

y para leer blancos podríamos llamar a algo como:

```
LeerPalabra(fich, blancos, True, CarsBlancos);
```

Este sería nuestro antiguo *LeerBlancos*, reconvertido según la idea que acabamos de exponer:

```
1     -- Lee un string formado por caracteres distintos a EOL
2     -- y a un separador. Se consideran separadores los
3     -- caracteres en sep (si no invertirsep) o aquellos
4     -- que no estan en sep (si invertirsep).

5     procedure LeerPalabra(fich: File_Type; s: out TipoString;
6                           invertirsep: Boolean; sep: String) is
7          fin: Boolean;
8          c: Character;
9     begin
10         s := NuevoString("");
11         fin := False;
```

```
        while not End_Of_File(fich) and not fin loop
             Look_Ahead(fich, c, fin);
             if not fin then
                  fin := EstaEn(sep, c);
                  if invertirsep then
                       fin := not fin;
                  end if;
             end if;
             if not fin then
                  Get(fich, c);
                  AppChar(s, c);
             end if;
        end loop;
   end;
```

Y ahora estamos en situación de leer una línea de texto. Podemos entrar en un bucle leyendo blancos y palabras alternativamente. Algo como...

```
    sonblancos := True;
    while not End_Of_File(fich) and not fin loop
         palabra := NuevoString("");
         if sonblancos then
              LeerPalabra(fich, palabra, False, CarsBlancos);
         else
              LeerPalabra(fich, palabra, True, CarsBlancos);
         end if;
         ...
         sonblancos := not sonblancos;
    end loop;
```

O lo que es lo mismo...

```
    sonblancos := True;
    while not End_Of_File(fich) and not fin loop
         palabra := NuevoString("");
         LeerPalabra(fich, palabra, not sonblancos, CarsBlancos);
         ...
         sonblancos := not sonblancos;
    end loop;
```

Y claro, puede que tengamos palabras vacías cuando encontramos una línea en blanco. En tal caso nuestro equivalente a saltar blancos encontraría el fin de línea y se detendría. Luego podríamos mejorar esto un poco y obtener el procedimiento que sigue para leer una línea:

```
    procedure LeerLinea(fich: File_Type; linea: out TipoLinea) is
         fin: Boolean;
         palabra: TipoString;
         sonblancos: Boolean;
         c: Character;

    begin
         linea := NuevaLinea;
         fin := False;
         sonblancos := False;
```

```
12          while not End_Of_File(fich) and not fin loop
13               palabra := NuevoString("");
14               LeerPalabra(fich, palabra, not sonblancos, CarsBlancos);
15               if LongString(palabra) /= 0 then
16                    AppPalabra(linea, palabra);
17               end if;
18               DestruirString(palabra);
19               look_ahead(fich, c, fin);
20               if fin  then
21                    Skip_Line(fich);
22               end if;
23               sonblancos := not sonblancos;
24          end loop;
25     end;
```

Una vez que hemos añadido una palabra a una línea hemos de destruir nuestra copia (la que hemos utilizado para leer la palabra). Recuerda que *AppPalabra* copiaba la palabra antes de añadirla. Intentamos que, aunque sean estructuras dinámicas, el comportamiento sea siempre similar a lo que pasaría si utilizásemos enteros o cualquier otro tipo estático. De ese modo se evitan sorpresas.

12.6. Textos

Una vez tenemos líneas podemos considerar un texto completo. Para nosotros el texto de un fichero será una serie de líneas. Además, puede ser interesante saber cuántas líneas tiene el texto (podría haberlo sido saber cuantas palabras tiene una línea pero, lo hecho, hecho está).

```
1      type TipoNodoLinea;
2      type TipoListaLinea is access all TipoNodoLinea;
3      type TipoNodoLinea is
4      record
5           linea: TipoLinea;
6           siguiente: TipoListaLinea;
7      end record;

9      type TipoTexto is
10     record
11          numlineas: Natural;
12          primera: TipoListaLinea;
13          ultima: TipoListaLinea;
14     end record;
```

Crear un texto es fácil.

```
1      function NuevoTexto return TipoTexto is
2           texto: TipoTexto;
3      begin
4           texto.primera := null;
5           texto.ultima := null;
6           texto.numlineas := 0;
7           return texto;
8      end;
```

Sabemos que tendremos que leer un texto a base de leer, una tras otra, sus líneas. Ya hemos hecho lo mismo con las palabras de una línea. Por lo tanto, además de las operaciones obvias para crear un texto y para destruirlo también vamos a querer otra, al menos, para añadir una línea a un texto.

```
1     -- Pone linea al final del texto. Como una linea puede ser
2     -- larga no hacemos ninguna copia. El texto se queda la linea.
3     procedure AppLinea(texto: in out TipoTexto; Linea: in out TipoLinea) is
4          pnodo: TipoListaLinea;
5     begin
6          pnodo := new TipoNodoLinea;
7          pnodo.linea := linea;      -- NO se hace copia!
8          pnodo.siguiente := null;
9          if texto.primera = null then
10              texto.primera := pnodo;
11         else
12              texto.ultima.siguiente := pnodo;
13         end if;
14         texto.ultima := pnodo;
15    end;
```

¡Cuidado aquí! Una línea podría ser muy larga. No queremos hacer una copia de toda la línea sólo para añadirla al texto (y que luego el procedimiento que lee el texto destruya su copia original de la línea). En lugar de eso vamos a hacer que *AppLinea* se quede la línea en propiedad y la añada tal cual está al texto. Eso quiere decir que el que llama a *AppLinea no puede* utilizar en absoluto la línea una vez que se la ha dado a *AppLinea*. ¡Ya no es suya tras la llamada! Ahora la línea forma parte del texto. Dado que esto es un poquito raro lo hemos detallado en los comentarios. Para eso están, para evitar sorpresas y aclarar cosas.

Además hemos puesto otro comentario en la misma línea que podría ser responsable de nuestras futuras desdichas, indicando que no hacemos copia de la línea.

Leer un texto es cuestión de hacer ahora con las líneas lo que hacíamos con las palabras. Está el pequeño detalle de que, en esta ocasión, debemos actualizar el número de líneas del texto y de que todas las líneas son iguales para nosotros (no hay ‘‘palabras’’ y ‘‘blancos’’). Pero... un momento. Tendremos que, en ocasiones, leer líneas hasta una línea que únicamente tenga un ‘‘`.`’’ (utilizada para marcar el final de la entrada). Mira el comando ‘‘`i`’’ del editor si no recuerdas esto.

En lugar de implementar dos operaciones, una para leer un texto de un fichero de datos y otra para leer un texto hasta una línea que sólo contenga un ‘‘`.`’’, vamos a implementar una única operación que se detenga cuando encuentra una línea cómo la que le indicamos. Este es el resultado.

```
1     procedure LeerTexto(fich: File_Type; texto: out TipoTexto; fin: String) is
2          linea: TipoLinea;
3          esfin: Boolean;
4     begin
5          texto := NuevoTexto;
6          esfin := False;
7          while not esfin loop
8               esfin := End_Of_File(fich);
9               if not esfin then
10                   LeerLinea(fich, linea);
11                   esfin := EsLineaFin(linea, fin);
12                   if esfin then
13                        DestruirLinea(linea);
14                   else
15                        AppLinea(texto, linea);
16                        texto.numlineas := texto.numlineas + 1;
17                   end if;
18              end if;
19         end loop;
20    end;
```

Hemos utilizado una función auxiliar para que nos diga si una línea consiste sólo en la palabra

indicada por el *string fin*. En tal caso destruimos dicha línea. En otro caso la incluimos en el texto.

Necesitaremos por lo menos dos operaciones más. Una para escribir un texto y otra para destruirlo. Podemos hacerlas ya. Esta es la primera:

```
1     procedure EscribirTexto(texto: in out TipoTexto) is
2           lista: TipoListaLinea;
3     begin
4           lista := texto.primera;
5           while lista /= null loop
6                 EscribirLinea(lista.linea);
7                 lista := lista.siguiente;
8           end loop;
9     end;
```

¿Empieza a resultar todo igual de mecánico y sencillo? Puede que sea largo, pero no es difícil. Lo único que puede hacer que deje de ser así es que olvidemos la sugerencia de hacer las cosas poco a poco, y compilarlas, y ejecutarlas, y probarlas cada vez. Cuando tengamos un error no queremos estar pensando dónde podrá estar. Queremos poder localizarlo escribiendo algunos *Put* en una zona pequeña de código (el nuevo que hemos añadido), y pasar rápido a otra cosa.

Esta es la segunda operación:

```
1     procedure DestruirTexto(texto: in out TipoTexto) is
2           nodo: TipoListaLinea;
3     begin
4           while texto.primera /= null loop
5                 nodo := texto.primera;
6                 texto.primera := nodo.siguiente;
7                 DestruirLinea(nodo.linea);
8                 Delete(nodo);
9           end loop;
10          texto.primera := null;
11          texto.ultima := null;
12    end;
```

Ahora estaría bien escribir un programa que lea el fichero de datos en un texto y lo escriba. Pero omitimos tal cosa. Como acabamos de decir un poco más arriba, sugerimos que siempre lo hagas, como costumbre. Cada vez que añadas un tipo o una operación nueva.

12.7. Comandos

Ha llegado el momento de permitir que el usuario del programa edite el texto. Sabemos que vamos a tener una serie de comandos. Además, cada comando puede tener una serie de argumentos. Por ejemplo, algunos reciben dos números de línea, otros una y otros ninguna. También tenemos otros comandos que reciben una palabra (el de editar) y otros en realidad reciben dos (el de cambiar una palabra por otra, aunque su sintaxis sea un poco rara, pero siguen siendo dos palabras).

Como sabemos que tenemos que implementar todos estos comandos, podemos definir un tipo de datos que sirva para manipular los comandos y luego ya nos preocuparemos de ir, uno por uno, implementando el código que se ocupe de ejecutarlos.

Lo único que tienen en común todos los comandos es algo que identifique de qué comando hablamos (lo llamaremos código) y el tipo y número de argumentos que utilizan (a esto lo llamaremos aridad). Estos podrían ser los códigos de nuestros comandos:

```
1     type TipoCodigo is (Insertar, Borrar, Imprimir, Cambiar,
2                 Editar, Guardar, Cual, Listar, Error);
```

Hemos incluido un código llamado *Error* (como hicimos en la calculadora). Resultará útil cuando el usuario se equivoque o queramos marcar un comando como uno erróneo.

En cuanto a la aridad de un comando, los tenemos sin argumentos, con un rango de números (dos números, uno o ninguno), con una palabra y con dos palabras como argumentos.

```
1     type TipoAridad is (NoArgs, Rango, Palabra, Palabras);
```

Aquí estamos siendo un poco astutos. Vamos a hacer que, cuando un comando que acepta números de línea sólo reciba uno, siga utilizando un rango (puede que con el mismo número como comienzo y final). Cuando leamos comandos ajustaremos el rango para que todo cuadre.

Para el caso en que tengamos un rango o dos palabras como argumentos necesitaremos un *array* para los argumentos. Estos son:

```
1     MaxArgs: constant Integer := 2;

3     type TipoArgs is array(1..MaxArgs) of Natural;
4     type TipoSArgs is array(1..MaxArgs) of TipoString;
```

Y este es el comando:

```
1     type TipoComando(aridad: TipoAridad := NoArgs) is
2     record
3          codigo: TipoCodigo;
4          case aridad is
5          when NoArgs =>
6               null;
7          when Rango =>
8               args: TipoArgs;
9          when Palabra =>
10              sarg: TipoString;
11         when Palabras =>
12              sargs: TipoSArgs;
13         end case;
14    end record;
```

Es un *record* con variantes. Todos los comandos tendrán el campo *aridad* para indicar que tipo/número de argumentos tienen y el campo *codigo* para indicar de qué comando se trata. El resto depende del valor del campo *aridad*.

Para leer un comando podemos leer una línea de texto y luego analizar dicha línea para ver qué comando tenemos entre manos. En función del comando podemos leer unos u otros argumentos.

Puesto que no lo hemos utilizado hasta ahora, y es un procedimiento útil, vamos a utilizar la operación *Get_Line* de *Ada.Text_IO* para leer una línea de texto. Los comandos sabemos qué longitud tienen (no mucha), por lo que no plantea problemas utilizar este procedimiento para este fin.

El código puede tener este aspecto:

```
1     procedure LeerComando(c: out TipoComando) is
2          buffer: String(1..MaxComando);
3          pri: Integer;
4          fin: Integer;
5          codigo: TipoCodigo;
6          ok: Boolean;

8     begin
9          Get_Line(buffer, fin);
10         pri := Saltar(buffer, buffer'First, fin, CarsBlancos);
```

```
12          if pri > fin then
13               c := (NoArgs, Error);
14          else
15               codigo := ParseCodigo(buffer(pri));
16               if codigo = Error then
17                    Put("Comando no reconocido");
18                    New_Line;
19               end if;
20               pri := pri + 1;

22               case codigo is
23               when Insertar..Imprimir =>
24                    ...
25               when Cambiar =>
26                    ...
27               when Editar =>
28                    ...
29               when others =>
30                    c := (NoArgs, codigo);
31               end case;
32          end if;
33     end;
```

La idea es que leemos en *buffer* una línea (hemos definido una constante *MaxComando* para la longitud máxima de un comando). Luego nos saltamos el espacio en blanco en el *string* que hemos leído. Esto lo hacemos llamando a un nuevo procedimiento *Saltar* al que le damos un *string*, una posición final y los caracteres que tiene que saltar. *Saltar* nos devuelve el índice en el *string* para el primer carácter, tras saltarse los que le hemos indicado. Así que si, tras saltar los blancos, *pri* es mayor que *fin*, entonces todo el *string* eran blancos. En ese caso creamos un comando que es un error.

Si tenemos algo en la línea aplicamos la misma idea. Llamamos a *ParseCodigo* (las funciones que analizan texto suelen siempre llamarse *parsers*). Dicha función nos devuelve un código para el comando o un código *Error* si era un comando no reconocido. Como los comandos son de una letra basta suministrar un carácter tras los blancos. Pero eso sí, incrementamos *pri*, puesto que dicha variable marca la posición en el *string* que es el principio de lo que nos falta por mirar.

Lo siguiente es ver qué tipo de comando tenemos y analizar el resto de la línea para ver sus argumentos. Veámoslo por partes.

Comandos como *Insertar* o *Imprimir* deben recibir un rango. Para no tener que programar por separado cada una de las posibilidades optamos por definir por omisión que los dos extremos del rango son *NoArg* (vamos, que no hay argumento definido). Luego llamamos a una función para que analice el resto del *string* y extraiga, o bien un número, o bien dos. Cada comando podrá ver si tiene uno o dos números como argumentos y actuar en consecuencia. Como podrían producirse errores al analizar el *string*, haremos que *ParseArgs* nos indique si hay algún problema, en cuyo caso nos olvidaremos del código que teníamos para el comando y lo definiremos como un error.

```
1     case codigo is
2     when Insertar..Imprimir =>
3          c := (Rango, codigo, (NoArg, NoArg));
4          ParseArgs(buffer(pri..fin), c.args, ok);
5          if not ok then
6               c := (NoArgs, Error);
7          end if;
8     when ...
```

Cuando el comando requiera cambiar una palabra por otra hacemos algo parecido. Por supuesto,

en este caso tendremos que utilizar otro procedimiento para analizar lo que queda del *string*. Además, vamos a considerar como un error que no se hayan indicado las dos palabras en el formato requerido. Todo el trabajo lo hará *ParseSargs*.

```
1     when Cambiar =>
2          c := (Palabras, codigo, (NuevoString(""), NuevoString("")));
3          ParseSargs(buffer(pri..fin), c.sargs);
4          if LongString(c.sargs(1)) = 0 or LongString(c.sargs(2)) = 0 then
5               Put("argumentos invalidos");
6               New_Line;
7               c := (NoArgs, Error);
8          end if;
```

Para el comando *Editar* necesitamos extraer una única palabra. Por lo demás es similar. El trabajo lo hará *ParseSarg*.

```
1     when Editar =>
2          c := (Palabra, codigo, NuevoString(""));
3          ParseSarg(buffer(pri..fin), c.sarg);
4          if LongString(c.sarg) = 0 then
5               Put("argumento invalido");
6               New_Line;
7               c := (NoArgs, Error);
8          end if;
```

12.8. Código y argumentos

Convertir un carácter a un código es fácil. Empecemos por ahí.

```
1     function ParseCodigo(c: Character) return TipoCodigo is
2          codigo: TipoCodigo;
3     begin
4          case c is
5          when 'd' =>
6               codigo := Borrar;
7          when 'p' =>
8               codigo := Imprimir;
9          when 'i' =>
10              codigo := Insertar;
11         when 's' =>
12              codigo := Cambiar;
13         when 'e' =>
14              codigo := Editar;
15         when 'w' =>
16              codigo := Guardar;
17         when 'f' =>
18              codigo := Cual;
19         when 'x' =>
20              codigo := Listar;
21         when others =>
22              codigo := Error;
23         end case;
24         return codigo;
25    end;
```

Extraer un único argumento (un *string*) del *string* que tenemos es fácil. Hay que saltar los blancos tras el comando y extraer el resto.

```
1     procedure ParseSArg(buffer: String; sarg: out TipoString) is
2          pri: Integer;
3     begin
4          pri := Saltar(buffer, buffer'First, buffer'Last, CarsBlancos);
5          if pri <= buffer'Last then
6               sarg := NuevoString(buffer(pri..buffer'Last));
7          else
8               sarg := NuevoString("");
9          end if;
10    end;
```

Extraer los argumentos cuando son dos palabras en el formato del comando *Cambiar* es un poco más delicado. Nosotros miramos cuál es el primer carácter tras los blancos y utilizamos ese carácter para separar una palabra de la siguiente. Nótese que este formato permite que las palabras que tenemos que cambiar contengan espacio en blanco. No obstante, el enunciado que teníamos no lo permite (Vamos, que esta sintaxis es un poco absurda en este caso). En cualquier caso, parece que a la gente que utiliza sistemas tipo UNIX esta sintaxis les gusta.

```
1     procedure ParseSArgs(buffer: String; sargs: out TipoSArgs) is
2          pri: Integer;
3          fin: Integer;
4          sep: Character;
5          i: Integer;
6     begin
7          pri := buffer'First;
8          if pri <= buffer'Last then
9               sep := buffer(pri);
10              pri := pri + 1;
11         else
12              sep := 'x';     -- cualquiera; esto es un error;
13         end if;
14         i := 1;
15         while i <= MaxArgs and pri <= buffer'Last loop
16              fin := SaltarOtros(buffer, pri, buffer'Last, "" & sep);
17              if pri < fin then
18                   sargs(i) := NuevoString(buffer(pri..fin-1));
19              else
20                   sargs(i) := NuevoString("");
21              end if;
22              pri := fin + 1;
23              i := i + 1;
24         end loop;
25    end;
```

Como se habrá podido ver, el procedimiento es capaz de leer un número indeterminado de palabras. Dado que tenemos que leer al menos dos, cuesta el mismo trabajo hacer un procedimiento que pueda leer cualquier número. De hecho, cuesta menos trabajo hacerlo así. Nos hemos permitido utilizar

```
          "" & sep
```

para convertir un carácter en un *String*. Ahora ya sabes hacer algo más con los *strings* de Ada. Puedes concatenar caracteres a *strings* y además puedes ver cómo de fácil resulta concatenar algo.

Para leer números vamos a utilizar *Saltar* para que nos diga qué trozo del *string* consiste en caracteres numéricos. Después podemos utilizar el atributo *Value* del tipo de datos *Integer* para convertir un *string* en un entero. (Sí, Ada también sabe hacer eso).

```
1     procedure ParseArgs(buffer: String; args: out TipoArgs; ok: out Boolean) is
2          pri: Integer;
3          fin: Integer;
4          i: Integer;

6     begin
7          pri := buffer'First;
8          i := 1;
9          while i <= MaxArgs and pri <= buffer'Last loop
10              pri := Saltar(buffer, pri, buffer'Last, CarsBlancos);

12              if pri <= buffer'Last then
13                   fin := Saltar(buffer, pri, buffer'last, "0123456789");
14              else
15                   fin := pri - 1;
16              end if;

18              if pri < fin then
19                   args(i) := Integer'Value(buffer(pri..fin-1));
20              else
21                   args(i) := NoArg;
22              end if;

24              pri := fin;
25              i := i + 1;
26         end loop;

28         if pri <= buffer'Last then
29              pri := Saltar(buffer, pri, buffer'Last, CarsBlancos);
30         end if;

32         ok := pri > buffer'Last;
33         if not ok then
34              Put("Argumentos invalidos");
35              New_Line;
36         end if;
37    end;
```

Nos faltan los procedimientos *Saltar* y *SaltarOtros*, que incluimos a continuación.

```
1     function Saltar(buffer: String; principio: Integer;
2                     fin: Integer; cars: String) return Integer is
3          terminado: Boolean;
4          pri: Integer;
5     begin
6          terminado := False;
7          pri := principio;
8          while pri <= fin and not terminado loop
9               if  EstaEn(cars, buffer(pri))  then
10                   pri := pri + 1;
11              else
12                   terminado := True;
13              end if;
14         end loop;
15         return pri;
16    end;
```

```
18    function SaltarOtros(buffer: String; principio: Integer;
19                    fin: Integer; cars: String) return Integer is
20         terminado: Boolean;
21         pri: Integer;
22    begin
23         terminado := False;
24         pri := principio;
25         while pri <= fin and not terminado loop
26              if  not EstaEn(cars, buffer(pri))  then
27                   pri := pri + 1;
28              else
29                   terminado := True;
30              end if;
31         end loop;
32         return pri;
33    end;
```

12.9. Editando el texto.

Estamos en situación de aplicar algunos comandos al texto. Por ejemplo, el comando para imprimir una o varias líneas basta con que itere sobre las líneas del texto e imprima las líneas seleccionadas por los argumentos del comando. Vamos a hacer que, si no hay especificados argumentos para la primera y última línea a que se refiere el comando, entonces el comando trabaje con todo el texto. Si se indica un único argumento haremos que el rango se refiera sólo a esa línea. Puesto que esto parece que lo tendremos que hacer con otros comandos, lo implementamos en un procedimiento *ArgsOmision* que defina los dos argumentos con los números de la primera y última línea con independencia de cuántos números haya escrito el usuario.

```
1     procedure ArgsOmision(texto: in out TipoTexto;
2                     arg1: in out Natural; arg2: in out Natural) is
3     begin
4          if arg2 = NoArg then
5               if arg1 = NoArg then
6                    arg1 := 1;
7                    arg2 := texto.numlineas;
8               else
9                    arg2 := arg1;
10              end if;
11         end if;
12    end;

14    procedure CmdImprimirTexto(texto: in out TipoTexto;
15                    arg1: in out Natural; arg2: in out Natural) is
16         i: Integer;
17         lista: TipoListaLinea;
18    begin
19         ArgsOmision(texto, arg1, arg2);
20         i := 1;
21         lista := texto.primera;
22         while lista /= null and i <= arg2 loop
23              if i >= arg1 then
24                   EscribirLinea(lista.linea);
25              end if;
26              lista := lista.siguiente;
27              i := i + 1;
28         end loop;
29    end;
```

Para borrar líneas podemos inventarnos una función *BorrarLinea* que borre una línea del texto y llamarla cuantas veces necesitemos para eliminar las líneas seleccionadas por los argumentos. Ha merecido la pena implementar *ArgsOmision*.

```
1     procedure CmdBorrarTexto(texto: in out TipoTexto;
2                     arg1: in out Natural; arg2: in out Natural) is
3     begin
4          ArgsOmision(texto, arg1, arg2);
5          for i in arg1..arg2 loop
6               BorrarLinea(texto, i);
7          end loop;
8     end;
```

Borrar una línea requiere localizarla, teniendo en cuenta que puede que el usuario indicase un número de línea inexistente.

```
1     procedure BorrarLinea(texto: in out TipoTexto; n: Natural) is
2          pnodo: TipoListaLinea;
3          panterior: TipoListaLinea;
4          i: Natural;
5     begin
6          panterior := null;
7          pnodo := texto.primera;
8          i := 1;
9          while pnodo /= null and i < n loop
10              pnodo := pnodo.siguiente;
11              panterior := pnodo;
12              i := i + 1;
13         end loop;

15         if pnodo = null then
16              Put("Linea " & Natural'Image(n) & " no encontrada");
17              New_Line;
18         else

20              if panterior = null then
21                   if texto.ultima = texto.primera then
22                        texto.ultima := null;
23                   end if;
24                   texto.primera := texto.primera.siguiente;
25              else
26                   panterior.siguiente := pnodo.siguiente;
27                   if texto.ultima = pnodo then
28                        texto.ultima := panterior;
29                   end if;
30              end if;

32              texto.numlineas := texto.numlineas - 1;
33              DestruirLinea(pnodo.linea);
34              Delete(pnodo);
35         end if;
36    end;
```

Insertar nuevas líneas requerirá leer un nuevo texto de la entrada, hasta la línea terminada por ‘‘.’’, y luego insertarlo en la posición adecuada. Ya que podemos borrar texto, podemos hacer que, cuando se pida insertar las líneas que se leen en un intervalo, lo que ocurra sea que primero se borran las líneas del intervalo y luego se insertan las nuevas. Suponiendo que tenemos una función *InsertarLineas*, podríamos escribir el procedimiento para este programa como sigue.

```
1     procedure CmdInsertarTexto(texto: in out TipoTexto;
2                     arg1: in out Natural; arg2: in out Natural) is
3          aux: TipoTexto;
4     begin
5          if arg1 /= NoArg and arg2 /= NoArg then
6               CmdBorrarTexto(texto, arg1, arg2);
7          else
8               if arg1 = NoArg then
9                    arg1 := texto.numlineas;
10              end if;
11         end if;
12         LeerTexto(Standard_Input, aux, ".");
13         InsertarLineas(texto, arg1, aux);     -- texto se queda con aux
14    end;
```

Para insertar las líneas hay distintos casos. Por ejemplo, que no haya líneas en el texto, que se inserten al principio, etc. Lo que haremos será tener en cuenta todos los casos especiales y luego escribir el código que inserta las nuevas líneas en el caso general. Igual que sucedía con insertar una línea, haremos que todo el texto que insertamos se lo quede el texto en el que lo insertamos, sin hacer copia alguna.

```
1     -- Inserta lineas en el texto sin copiarlas.
2     procedure InsertarLineas(texto: in out TipoTexto;
3                     pos: Natural; lineas: in out TipoTexto) is
4          panterior: TipoListaLinea;
5     begin
6          -- Primero consideramos los casos especiales y luego
7          -- implementamos el general ignorando estos primeros.
8          if texto.numlineas = 0 then
9               texto := lineas;
10         elsif lineas.numlineas = 0 then
11              null;

13         elsif pos <= 1 then
14              -- insertar al principio
15              lineas.ultima.siguiente := texto.primera;
16              texto.primera := lineas.primera;

18         elsif pos > texto.numlineas then
19              -- insertar al final
20              texto.ultima.siguiente := lineas.primera;
21              texto.ultima := lineas.ultima;
22         else

24              -- insertar en medio; panterior no puede hacerse null.
25              -- lo hacemos avanzar hasta que sea en nodo anterior
26              -- al punto de insercion.
27              panterior := texto.primera;
28              for i in 1..pos-2 loop
29                   panterior := panterior.siguiente;
30              end loop;
31              lineas.ultima.siguiente := panterior.siguiente;
32              panterior.siguiente := lineas.primera;
33         end if;
34         texto.numlineas := texto.numlineas + lineas.numlineas;
35         lineas.primera := null;
36         lineas.ultima := null;
37    end;
```

Cambiar una palabra por otra en el texto requiere recorrerlo y comparar cada palabra con la que

queremos cambiar. Para cambiarla copiaremos el texto de la nueva palabra sobre la palabra antigua.

```
1     procedure CambiarPalabra(linea: in out TipoLinea;
2                     arg1: TipoString; arg2: TipoString) is
3          lista: TipoListaPalabra;
4     begin
5          lista := linea.primera;
6          while lista /= null  loop
7               if IgualString(lista.palabra, arg1) then
8                    CopiarString(lista.palabra, arg2);
9               end if;
10              lista := lista.siguiente;
11         end loop;
12    end;

14    procedure CmdCambiarTexto(texto: in out TipoTexto;
15                    arg1: TipoString; arg2: TipoString) is
16         lista: TipoListaLinea;
17    begin
18         lista := texto.primera;
19         while lista /= null  loop
20              CambiarPalabra(lista.linea, arg1, arg2);
21              lista := lista.siguiente;
22         end loop;
23    end;
```

12.10. Un editor de un sólo fichero.

Ahora estamos en situación de tener un editor para un único fichero. Tenemos que leer el fichero en un texto y luego leer un comando tras otro, aplicándolos al texto.

Aunque por el momento sólo tenemos un fichero sabemos que tendremos que editar varios a la vez. Por tanto tiene sentido definir un tipo de datos para una edición, que al menos agrupe el texto de un fichero con el nombre del fichero y cualquier otra cosa que podamos necesitar (por ejemplo, una indicación de si hemos modificado el fichero con comandos de edición o no).

```
1     type TipoEstadoEdicion is (Limpio, Sucio);

3     type TipoEdicion is
4     record
5          nombre: TipoString;
6          texto: TipoTexto;
7          estado: TipoEstadoEdicion;
8     end record;
```

Decimos que un fichero está limpio si no hemos hecho cambios y sucio si los hemos hecho pero no hemos escrito el contenido del fichero tras los cambios.

En cualquier caso, nuestro programa debe simplemente establecer una nueva edición para el fichero `datos.txt` (por el momento) y aplicar los comandos oportunos al texto. Podría quedar entonces como sigue:

```
        edicion: TipoEdicion;
        comando: TipoComando;
    begin
        NuevaEdicion("datos.txt", edicion);
        while not End_Of_File loop
            LeerComando(comando);
            ProcesarComando(edicion, comando);
            DestruirComando(comando);
        end loop;
        EscribirTexto(edicion.texto);
    end;
```

El procedimiento *NuevaEdicion* crea una nueva edición, leyendo el fichero cuyo nombre se indica y leyendo su contenido con ayuda de *LeerTexto*:

```
    procedure NuevaEdicion(nombre: String; edicion: out TipoEdicion) is
        fich: File_Type;
    begin
        edicion.nombre := NuevoString(nombre);
        edicion.estado := Limpio;
        Open(fich, In_File, nombre);
        LeerTexto(fich, edicion.texto, "");
        Close(fich);
        Put(nombre & ": ");
        Put(Integer'Image(edicion.texto.numlineas) & " lineas");
        New_Line;
    end;
```

Leer_Comando ya lo teníamos programado y ahora nos falta procesarlo y destruirlo. Recuerda que un comando es una estructura dinámica y, como de costumbre ya, hay que liberar toda la memoria que hemos pedido.

Para procesar un comando todo depende de qué comando tenemos que procesar.

```
    procedure ProcesarComando(edicion: in out TipoEdicion; c: in out TipoComando) is
    begin
        case c.codigo is
        when Insertar =>
            CmdInsertarTexto(edicion.texto, c.args(1), c.args(2));
            edicion.estado := Sucio;
        when Borrar =>
            CmdBorrarTexto(edicion.texto, c.args(1), c.args(2));
            edicion.estado := Sucio;
        when Imprimir =>
            CmdImprimirTexto(edicion.texto, c.args(1), c.args(2));
        when Cambiar =>
            CmdCambiarTexto(edicion.texto, c.sargs(1), c.sargs(2));
            edicion.estado := Sucio;
        when others =>
            -- aun no tenemos comandos para mas de un fichero.
            null;
        end case;
    end;
```

Destruir el comando es sencillo. Basta destruir los argumentos, caso de ser necesario (en función del tipo de comando).

```
1     procedure DestruirComando(c: in out TipoComando) is
2     begin
3          case c.aridad is
4          when Palabras =>
5               DestruirString(c.sargs(1));
6               DestruirString(c.sargs(2));
7          when Palabra =>
8               DestruirString(c.sarg);
9          when others =>
10              null;
11         end case;
12    end;
```

¡Se nos ha olvidado! Tenemos que implementar el comando para guardar los cambios hechos en el fichero. No hay problema. Necesitamos un procedimiento que implemente el comando, como de costumbre:

```
1     procedure CmdGuardar(edicion: in out TipoEdicion) is
2     begin
3          if edicion.estado = Sucio then
4               EscribirTexto(edicion.texto);
5               edicion.estado := Limpio;
6          end if;
7     end;
```

Estamos escribiendo el resultado de la edición en la salida estándar, pero escribirlo en un fichero sería igual de sencillo. Eso sí, sólo lo hacemos si habíamos hecho cambios. En otro caso no hay nada que guardar.

Falta contemplar el comando en *ProcesarComando*:

```
1          when Guardar =>
2               CmdGuardar(edicion);
```

Y con esto tenemos todo lo necesario para editar un único fichero, llamado `datos.txt`. Nos resta generalizar el editor para que pueda manipular más de un fichero. Si el código y las estructuras de datos están limpios, no debiera ser difícil. En otro caso sería mejor rehacer todo lo que pueda haber quedado mal hasta que resulte sencillo hacer el cambio.

12.11. Edición de múltiples ficheros

Lo único que necesitamos es mantener una lista de ediciones y aplicar los comandos de edición a aquella que podamos tener como edición actual. Una lista de ediciones se define como cabría esperar:

```
1     type TipoNodoEdicion;
2     type TipoListaEdicion is access all TipoNodoEdicion;
3     type TipoNodoEdicion is
4     record
5          edicion: TipoEdicion;
6          siguiente: TipoListaEdicion;
7     end record;
```

El programa principal va a mantener ahora una lista de ediciones y, también, un puntero a la edición actual. Este último es preciso puesto que muchos comandos (todos los implementados hasta el momento) se refieren a la edición en curso. Podemos modificar primero el programa principal y luego implementar el resto de operaciones según resulte necesario.

```
1          ediciones: TipoListaEdicion;
2          pedicion: TipoListaEdicion;
3          comando: TipoComando;
4     begin
5          CmdEditar(ediciones, pedicion, "datos.txt");
6          while not End_Of_File loop
7               LeerComando(comando);
8               ProcesarComando(ediciones, pedicion, comando);
9               DestruirComando(comando);
10         end loop;
11         EscribirTexto(pedicion.edicion.texto);
12    end;
```

Ediciones será la lista de ediciones y *pedicion* será el puntero a la edición actual. Vamos a hacer que el editor empiece por editar `datos.txt` justo al comenzar (aunque sólo sea para probar). La idea es que *CmdEditar* implementa el comando ‘‘`e`’’, encargado de editar un nuevo fichero. Tendremos que pasarle tanto la lista de ediciones como el puntero a la edición actual, ambos por referencia dado que ambas cosas pueden cambiar. Por esa misma razón se ve claro ya desde este momento que *ProcesarComando* debe recibir también ambas cosas (y también por referencia). Si esto no se viese no hay que preocuparse; la verdad es obstinada y acaba por imponerse.

El procedimiento *CmdEditar* es el equivalente a insertar una nueva edición en la lista de ediciones.

```
1     procedure CmdEditar(ediciones: in out TipoListaEdicion;
2               pedicion: in out TipoListaEdicion;
3               arg: String) is
4          lista: TipoListaEdicion;
5     begin
6          lista := ediciones;
7          pedicion := null;
8          while lista /= null and pedicion = null loop
9               if AdaString(lista.edicion.nombre) = arg then
10                   pedicion := lista;
11              else
12                   lista := lista.siguiente;
13              end if;
14         end loop;
15         if pedicion = null then
16              pedicion := new TipoNodoEdicion;
17              NuevaEdicion(arg, pedicion.edicion);
18              pedicion.siguiente := ediciones;
19              ediciones := pedicion;
20         end if;
21    end;
```

Un detalle aquí es que si ya estamos editando un fichero no debemos volver a crear una edición para el mismo. Basta con hacer que dicha edición sea la edición actual (esto es, que *pedicion* apunte a dicha edición). Así pues recorremos la lista de ediciones y, caso de no encontrar ninguna para el nombre de fichero indicado, creamos una nueva edición que enlazamos por el final a la lista.

Podemos ya rellenar la rama para este comando en *ProcesarComando* (además de cambiar sus argumentos como ya hemos dicho antes).

```
1     procedure ProcesarComando( ediciones: in out TipoListaEdicion;
2                     pedicion: in out TipoListaEdicion;
3                     c: in out TipoComando) is
4     begin
5          case c.codigo is
6          ...
7          when Editar =>
8               CmdEditar(ediciones, pedicion, AdaString(c.sarg));
9          ...
10         end case;
11    end;
```

Dado que tenemos ahora un puntero a la edición actual debemos cambiar más cosas en *ProcesarComando*. En particular hay que hacer que todas las llamadas a procedimientos que implementan comandos utilicen *pedicion.edicion* en lugar de utilizar *edicion* como hacían antes. Por ejemplo, el caso para el comando imprimir sería ahora:

```
1     when Imprimir =>
2          CmdImprimirTexto(pedicion.edicion.texto, c.args(1), c.args(2));
```

Puesto que podemos editar múltiples ficheros necesitamos poder preguntar cuál estamos editando ahora. Eso es tarea del comando ‘‘`f`’’ (representado por *Cual* en el *TipoCodigo*). Este es muy fácil:

```
1     procedure CmdCual(pedicion: in out TipoListaEdicion; imprestado: Boolean) is
2     begin
3          Put(AdaString(pedicion.edicion.nombre));
4          if imprestado and pedicion.edicion.estado = Sucio then
5               Put(" sucio");
6          end if;
7          New_Line;
8     end;
```

Como se verá lo hemos dotado de un segundo parámetro que, cuando es cierto, hace que el procedimiento indique también si el fichero tiene cambios sin grabar.

Para listar todas las ediciones (tarea del comando ‘‘`x`’’) podemos utilizar este comando, llamándolo para cada edición.

```
1     procedure CmdListar(ediciones: in out TipoListaEdicion) is
2          lista: TipoListaEdicion;
3     begin
4          lista := ediciones;
5          while lista /= null loop
6               CmdCual(lista, True);
7               lista := lista.siguiente;
8          end loop;
9     end;
```

12.12. Terminado

Tenemos nuestro flamante editor.

editor.adb

```
1     --
2     -- Editar la entrada
3     --
4     with Ada.Text_IO;
5     use Ada.Text_IO;
```

```
7     with Ada.Unchecked_Deallocation;

9     procedure Editor is

11          CarsBlancos: constant String := " " & ASCII.HT;

13          -- Los TipoString crecen de Incr en Incr caracteres
14          -- al ir llamando a AppChar
15          Incr: constant Integer := 32;

17          MaxComando: constant Integer := 200;
18          MaxArgs: constant Integer := 2;
19          NoArg: constant := 0;

21          type TipoCodigo is (Insertar, Borrar, Imprimir, Cambiar,
22                    Editar, Guardar, Cual, Listar, Error);

24          subtype TipoCodigoArgs is TipoCodigo range Insertar..Imprimir;

26          type TipoAridad is (NoArgs, Rango, Palabra, Palabras);

28          type TipoPtrString is access all String;

30          type TipoString is
31          record
32               cars: TipoPtrString;
33               long: Natural;
34               max: Natural;
35          end record;

37          type TipoArgs is array(1..MaxArgs) of Natural;
38          type TipoSArgs is array(1..MaxArgs) of TipoString;

40          type TipoComando(aridad: TipoAridad := NoArgs) is
41          record
42               codigo: TipoCodigo;
43               case aridad is
44               when NoArgs =>
45                    null;
46               when Rango =>
47                    args: TipoArgs;
48               when Palabra =>
49                    sarg: TipoString;
50               when Palabras =>
51                    sargs: TipoSArgs;
52               end case;
53          end record;

55          type TipoNodoPalabra;
56          type TipoListaPalabra is access all TipoNodoPalabra;
57          type TipoNodoPalabra is
58          record
59               palabra: TipoString;
60               siguiente: TipoListaPalabra;
61          end record;
```

```
63          type TipoLinea is
64          record
65               primera: TipoListaPalabra;
66               ultima: TipoListaPalabra;
67          end record;

69          type TipoNodoLinea;
70          type TipoListaLinea is access all TipoNodoLinea;
71          type TipoNodoLinea is
72          record
73               linea: TipoLinea;
74               siguiente: TipoListaLinea;
75          end record;

77          type TipoTexto is
78          record
79               numlineas: Natural;
80               primera: TipoListaLinea;
81               ultima: TipoListaLinea;
82          end record;

84          type TipoEstadoEdicion is (Limpio, Sucio);

86          type TipoEdicion is
87          record
88               nombre: TipoString;
89               texto: TipoTexto;
90               estado: TipoEstadoEdicion;
91          end record;

93          type TipoNodoEdicion;
94          type TipoListaEdicion is access all TipoNodoEdicion;
95          type TipoNodoEdicion is
96          record
97               edicion: TipoEdicion;
98               siguiente: TipoListaEdicion;
99          end record;

101         package IntIO is new Integer_IO(Integer);
102         use IntIO;

104         procedure Delete is new Ada.Unchecked_Deallocation(String, TipoPtrString);
105         procedure Delete is new Ada.Unchecked_Deallocation(TipoNodoPalabra, TipoListaPalabra);
106         procedure Delete is new Ada.Unchecked_Deallocation(TipoNodoLinea, TipoListaLinea);

108         function NuevoString(ini: String)  return TipoString is
109              s: TipoString;
110         begin
111              s.cars := new String(1..ini'Length);
112              s.long := ini'Length;
113              s.max := ini'Length;
114              s.cars(1..s.long) := ini;
115              return s;
116         end;
```

```
118         function LongString(s: TipoString) return Natural is
119         begin
120              return s.long;
121         end;

123         function AdaString(s: TipoString) return String is
124         begin
125              if s.long = 0 then
126                   return "";
127              else
128                   return s.cars(1..s.long);
129              end if;
130         end;

132         function IgualString(s1: TipoString; s2: TipoString) return Boolean is
133         begin
134              if s1.long /= s2.long then
135                   return False;
136              else
137                   return s1.cars(1..s1.long) = s2.cars(1..s2.long);
138              end if;
139         end;

141         procedure DestruirString(s: in out TipoString) is
142         begin
143              if s.cars /= null then
144                   Delete(s.cars);
145                   s.cars := null;
146                   s.max := 0;
147              end if;
148         end;

150         procedure CopiarString(dest: in out TipoString; orig: TipoString) is
151         begin
152              if dest.max < orig.long then
153                   Delete(dest.cars);
154                   dest.cars := new String(1..orig.long);
155                   dest.max := orig.long;
156              end if;
157              dest.cars(1..orig.long) := orig.cars.all;
158              dest.long := orig.long;
159         end;

161         procedure CrecerString(dest: in out TipoString; cuanto: Positive) is
162              paux: TipoPtrString;
163         begin
164              paux := dest.cars;
165              dest.cars := new String(1..dest.long + Incr);
166              dest.max := dest.long + Incr;
167              dest.cars(1..dest.long) := paux(1..dest.long);
168              Delete(paux);
169         end;
```

```
171         procedure AppChar(dest: in out TipoString; c: Character) is
172         begin
173              if dest.max < dest.long + 1 then
174                   CrecerString(dest, Incr);
175              end if;
176              dest.long := dest.long + 1;
177              dest.cars(dest.long) := c;
178         end;

180         procedure AppString(dest: in out TipoString; orig: TipoString) is
181              nuevolong: Natural;
182         begin
183              nuevolong := dest.long + orig.long;
184              if dest.max < nuevolong then
185                   CrecerString(dest, orig.long);
186              end if;
187              dest.cars(dest.long+1..nuevolong) := orig.cars.all;
188              dest.long := nuevolong;
189         end;

191         function EstaEn(s: String; c: Character) return Boolean is
192              esta: Boolean;
193              pos: Integer;
194         begin
195              esta := False;
196              pos := s'First;
197              while pos <= s'Last and not esta loop
198                   esta := s(pos) = c;
199                   pos := pos + 1;
200              end loop;
201              return esta;
202         end;

204         -- Lee un string formado por caracteres distintos a EOL
205         -- y a un separador. Se consideran separadores los
206         -- caracteres en sep (si no invertirsep) o aquellos
207         -- que no estan en sep (si invertirsep).
208         procedure LeerPalabra(fich: File_Type; s: out TipoString;
209                               invertirsep: Boolean; sep: String) is
210              fin: Boolean;
211              c: Character;
212         begin
213              s := NuevoString("");
214              fin := False;
215              while not End_Of_File(fich) and not fin loop
216                   Look_Ahead(fich, c, fin);
217                   if not fin then
218                        fin := EstaEn(sep, c);
219                        if invertirsep then
220                             fin := not fin;
221                        end if;
222                   end if;
223                   if not fin then
224                        Get(fich, c);
225                        AppChar(s, c);
226                   end if;
227              end loop;
228         end;
```

```
230         function NuevaLinea return TipoLinea is
231              nl: TipoLinea;
232         begin
233              nl.primera := null;
234              nl.ultima := null;
235              return nl;
236         end;

238         procedure AppPalabra(linea: in out TipoLinea; palabra: TipoString) is
239              pnodo: TipoListaPalabra;
240         begin
241              pnodo := new TipoNodoPalabra;
242              pnodo.palabra := NuevoString(AdaString(palabra));
243              pnodo.siguiente := null;
244              if linea.primera = null then
245                   linea.primera := pnodo;
246              else
247                   linea.ultima.siguiente := pnodo;
248              end if;
249              linea.ultima := pnodo;
250         end;

252         procedure EscribirLinea(linea: TipoLinea) is
253              lista: TipoListaPalabra;
254         begin
255              lista := linea.primera;
256              while lista /= null loop
257                   Put(AdaString(lista.palabra));
258                   lista := lista.siguiente;
259              end loop;
260              New_Line;
261         end;

263         procedure LeerLinea(fich: File_Type; linea: out TipoLinea) is
264              fin: Boolean;
265              palabra: TipoString;
266              sonblancos: Boolean;
267              c: Character;

269         begin
270              linea := NuevaLinea;
271              fin := False;
272              sonblancos := False;

274              while not End_Of_File(fich) and not fin loop
275                   palabra := NuevoString("");
276                   LeerPalabra(fich, palabra, not sonblancos, CarsBlancos);
277                   if LongString(palabra) /= 0 then
278                        AppPalabra(linea, palabra);
279                   end if;
280                   DestruirString(palabra);
281                   look_ahead(fich, c, fin);
282                   if fin  then
283                        Skip_Line(fich);
284                   end if;
285                   sonblancos := not sonblancos;
286              end loop;
287         end;
```

```
289         procedure DestruirLinea(linea: in out TipoLinea) is
290              nodo: TipoListaPalabra;
291         begin
292              while linea.primera /= null loop
293                   nodo := linea.primera;
294                   linea.primera := nodo.siguiente;
295                   DestruirString(nodo.palabra);
296                   Delete(nodo);
297              end loop;
298              linea.primera := null;
299              linea.ultima := null;
300         end;

302         function NuevoTexto return TipoTexto is
303              texto: TipoTexto;
304         begin
305              texto.primera := null;
306              texto.ultima := null;
307              texto.numlineas := 0;
308              return texto;
309         end;

311         -- Pone linea al final del texto. Como una linea puede ser
312         -- larga no hacemos ninguna copia. El texto se queda la linea.
313         procedure AppLinea(texto: in out TipoTexto; Linea: in out TipoLinea) is
314              pnodo: TipoListaLinea;
315         begin
316              pnodo := new TipoNodoLinea;
317              pnodo.linea := linea;           -- NO se hace copia!
318              pnodo.siguiente := null;
319              if texto.primera = null then
320                   texto.primera := pnodo;
321              else
322                   texto.ultima.siguiente := pnodo;
323              end if;
324              texto.ultima := pnodo;
325         end;

327         procedure BorrarLinea(texto: in out TipoTexto; n: Natural) is
328              pnodo: TipoListaLinea;
329              panterior: TipoListaLinea;
330              i: Natural;
331         begin
332              panterior := null;
333              pnodo := texto.primera;
334              i := 1;
335              while pnodo /= null and i < n loop
336                   pnodo := pnodo.siguiente;
337                   panterior := pnodo;
338                   i := i + 1;
339              end loop;

341              if pnodo = null then
342                   Put("Linea " & Natural'Image(n) & " no encontrada");
343                   New_Line;
344              else
```

```
346                 if panterior = null then
347                      if texto.ultima = texto.primera then
348                           texto.ultima := null;
349                      end if;
350                      texto.primera := texto.primera.siguiente;
351                 else
352                      panterior.siguiente := pnodo.siguiente;
353                      if texto.ultima = pnodo then
354                           texto.ultima := panterior;
355                      end if;
356                 end if;

358                 texto.numlineas := texto.numlineas - 1;
359                 DestruirLinea(pnodo.linea);
360                 Delete(pnodo);
361            end if;
362       end;

364       -- Inserta lineas en el texto sin copiarlas.
365       procedure InsertarLineas(texto: in out TipoTexto;
366                      pos: Natural; lineas: in out TipoTexto) is
367            panterior: TipoListaLinea;
368       begin
369            -- Primero consideramos los casos especiales y luego
370            -- implementamos el general ignorando estos primeros.
371            if texto.numlineas = 0 then
372                 texto := lineas;
373            elsif lineas.numlineas = 0 then
374                 null;

376            elsif pos <= 1 then
377                 -- insertar al principio
378                 lineas.ultima.siguiente := texto.primera;
379                 texto.primera := lineas.primera;

381            elsif pos > texto.numlineas then
382                 -- insertar al final
383                 texto.ultima.siguiente := lineas.primera;
384                 texto.ultima := lineas.ultima;
385            else

387                 -- insertar en medio; panterior no puede hacerse null.
388                 -- lo hacemos avanzar hasta que sea en nodo anterior
389                 -- al punto de insercion.
390                 panterior := texto.primera;
391                 for i in 1..pos-2 loop
392                      panterior := panterior.siguiente;
393                 end loop;
394                 lineas.ultima.siguiente := panterior.siguiente;
395                 panterior.siguiente := lineas.primera;
396            end if;
397            texto.numlineas := texto.numlineas + lineas.numlineas;
398            lineas.primera := null;
399            lineas.ultima := null;
400       end;
```

```
402         function EsLineaFin(linea: TipoLinea; fin: String) return Boolean is
403              loes: Boolean;
404         begin
405              if linea.primera /= null then
406                   loes := linea.primera.siguiente = null and
407                        AdaString(linea.primera.palabra) = fin;
408              else
409                   loes := False;
410              end if;
411              return loes;
412         end;

414         procedure LeerTexto(fich: File_Type; texto: out TipoTexto; fin: String) is
415              linea: TipoLinea;
416              esfin: Boolean;
417         begin
418              texto := NuevoTexto;
419              esfin := False;
420              while not esfin loop
421                   esfin := End_Of_File(fich);
422                   if not esfin then
423                        LeerLinea(fich, linea);
424                        esfin := EsLineaFin(linea, fin);
425                        if esfin then
426                             DestruirLinea(linea);
427                        else
428                             AppLinea(texto, linea);
429                             texto.numlineas := texto.numlineas + 1;
430                        end if;
431                   end if;
432              end loop;
433         end;

435         procedure EscribirTexto(texto: in out TipoTexto) is
436              lista: TipoListaLinea;
437         begin
438              lista := texto.primera;
439              while lista /= null loop
440                   EscribirLinea(lista.linea);
441                   lista := lista.siguiente;
442              end loop;
443         end;

445         procedure DestruirTexto(texto: in out TipoTexto) is
446              nodo: TipoListaLinea;
447         begin
448              while texto.primera /= null loop
449                   nodo := texto.primera;
450                   texto.primera := nodo.siguiente;
451                   DestruirLinea(nodo.linea);
452                   Delete(nodo);
453              end loop;
454              texto.primera := null;
455              texto.ultima := null;
456         end;
```

```
458         procedure NuevaEdicion(nombre: String; edicion: out TipoEdicion) is
459              fich: File_Type;
460         begin
461              edicion.nombre := NuevoString(nombre);
462              edicion.estado := Limpio;
463              Open(fich, In_File, nombre);
464              LeerTexto(fich, edicion.texto, "");
465              Close(fich);
466              Put(nombre & ": ");
467              Put(Integer'Image(edicion.texto.numlineas) & " lineas");
468              New_Line;
469         end;

471         procedure DestruirComando(c: in out TipoComando) is
472         begin
473              case c.aridad is
474              when Palabras =>
475                   DestruirString(c.sargs(1));
476                   DestruirString(c.sargs(2));
477              when Palabra =>
478                   DestruirString(c.sarg);
479              when others =>
480                   null;
481              end case;
482         end;

484         function Saltar(buffer: String; principio: Integer;
485                         fin: Integer; cars: String) return Integer is
486              terminado: Boolean;
487              pri: Integer;
488         begin
489              terminado := False;
490              pri := principio;
491              while pri <= fin and not terminado loop
492                   if  EstaEn(cars, buffer(pri))  then
493                        pri := pri + 1;
494                   else
495                        terminado := True;
496                   end if;
497              end loop;
498              return pri;
499         end;

501         function SaltarOtros(buffer: String; principio: Integer;
502                         fin: Integer; cars: String) return Integer is
503              terminado: Boolean;
504              pri: Integer;
```

```
505         begin
506              terminado := False;
507              pri := principio;
508              while pri <= fin and not terminado loop
509                   if  not EstaEn(cars, buffer(pri))  then
510                        pri := pri + 1;
511                   else
512                        terminado := True;
513                   end if;
514              end loop;
515              return pri;
516         end;

518         procedure ParseArgs(buffer: String; args: out TipoArgs; ok: out Boolean) is
519              pri: Integer;
520              fin: Integer;
521              i: Integer;
522         begin
523              pri := buffer'First;
524              i := 1;
525              while i <= MaxArgs and pri <= buffer'Last loop
526                   pri := Saltar(buffer, pri, buffer'Last, CarsBlancos);
527                   if pri <= buffer'Last then
528                        fin := Saltar(buffer, pri, buffer'last, "0123456789");
529                   else
530                        fin := pri - 1;
531                   end if;
532                   if pri < fin then
533                        args(i) := Integer'Value(buffer(pri..fin-1));
534                   else
535                        args(i) := NoArg;
536                   end if;
537                   pri := fin;
538                   i := i + 1;
539              end loop;
540              if pri <= buffer'Last then
541                   pri := Saltar(buffer, pri, buffer'Last, CarsBlancos);
542              end if;
543              ok := pri > buffer'Last;
544              if not ok then
545                   Put("Argumentos invalidos");
546                   New_Line;
547              end if;
548         end;

550         procedure ParseSArg(buffer: String; sarg: out TipoString) is
551              pri: Integer;
552         begin
553              pri := Saltar(buffer, buffer'First, buffer'Last, CarsBlancos);
554              if pri <= buffer'Last then
555                   sarg := NuevoString(buffer(pri..buffer'Last));
556              else
557                   sarg := NuevoString("");
558              end if;
559         end;
```

```
561         procedure ParseSArgs(buffer: String; sargs: out TipoSArgs) is
562              pri: Integer;
563              fin: Integer;
564              sep: Character;
565              i: Integer;
566         begin
567              pri := buffer'First;
568              if pri <= buffer'Last then
569                   sep := buffer(pri);
570                   pri := pri + 1;
571              else
572                   sep := 'x';     -- cualquiera; esto es un error;
573              end if;
574              i := 1;
575              while i <= MaxArgs and pri <= buffer'Last loop
576                   fin := SaltarOtros(buffer, pri, buffer'Last, "" & sep);
577                   if pri < fin then
578                        sargs(i) := NuevoString(buffer(pri..fin-1));
579                   else
580                        sargs(i) := NuevoString("");
581                   end if;
582                   pri := fin + 1;
583                   i := i + 1;
584              end loop;
585         end;

587         function ParseCodigo(c: Character) return TipoCodigo is
588              codigo: TipoCodigo;
589         begin
590              case c is
591              when 'd' =>
592                   codigo := Borrar;
593              when 'p' =>
594                   codigo := Imprimir;
595              when 'i' =>
596                   codigo := Insertar;
597              when 's' =>
598                   codigo := Cambiar;

599              when 'e' =>
600                   codigo := Editar;
601              when 'w' =>
602                   codigo := Guardar;
603              when 'f' =>
604                   codigo := Cual;
605              when 'x' =>
606                   codigo := Listar;
607              when others =>
608                   codigo := Error;
609              end case;
610              return codigo;
611         end;
```

```
613         procedure LeerComando(c: out TipoComando) is
614              buffer: String(1..MaxComando);
615              pri: Integer;
616              fin: Integer;
617              codigo: TipoCodigo;
618              ok: Boolean;
619         begin
620              Get_Line(buffer, fin);
621              pri := Saltar(buffer, buffer'First, fin, CarsBlancos);
622              if pri > fin then
623                   c := (NoArgs, Error);
624              else
625                   codigo := ParseCodigo(buffer(pri));
626                   if codigo = Error then
627                        Put("Comando no reconocido");
628                        New_Line;
629                   end if;
630                   pri := pri + 1;
631                   case codigo is
632                   when TipoCodigoArgs =>
633                        c := (Rango, codigo, (NoArg, NoArg));
634                        ParseArgs(buffer(pri..fin), c.args, ok);
635                        if not ok then
636                             c := (NoArgs, Error);
637                        end if;

638                   when Cambiar =>
639                        c := (Palabras, codigo, (NuevoString(""), NuevoString("")));
640                        ParseSargs(buffer(pri..fin), c.sargs);
641                        if LongString(c.sargs(1)) = 0 or LongString(c.sargs(2)) = 0 then
642                             Put("argumentos invalidos");
643                             New_Line;
644                             DestruirComando(c);
645                             c := (NoArgs, Error);
646                        end if;

647                   when Editar =>
648                        c := (Palabra, codigo, NuevoString(""));
649                        ParseSarg(buffer(pri..fin), c.sarg);
650                        if LongString(c.sarg) = 0 then
651                             Put("argumento invalido");
652                             New_Line;
653                             DestruirComando(c);
654                             c := (NoArgs, Error);
655                        end if;

656                   when others =>
657                        c := (NoArgs, codigo);
658                   end case;
659              end if;
660         end;
```

```
662         procedure ArgsOmision(texto: in out TipoTexto;
663                               arg1: in out Natural; arg2: in out Natural) is
664         begin
665              if arg2 = NoArg then
666                   if arg1 = NoArg then
667                        arg1 := 1;
668                        arg2 := texto.numlineas;
669                   else
670                        arg2 := arg1;
671                   end if;
672              end if;
673         end;

675         procedure CmdImprimirTexto(texto: in out TipoTexto;
676                               arg1: in out Natural; arg2: in out Natural) is
677              i: Integer;
678              lista: TipoListaLinea;
679         begin
680              ArgsOmision(texto, arg1, arg2);
681              i := 1;
682              lista := texto.primera;
683              while lista /= null and i <= arg2 loop
684                   if i >= arg1 then
685                        EscribirLinea(lista.linea);
686                   end if;
687                   lista := lista.siguiente;
688                   i := i + 1;
689              end loop;
690         end;

692         procedure CmdBorrarTexto(texto: in out TipoTexto;
693                               arg1: in out Natural; arg2: in out Natural) is
694         begin
695              ArgsOmision(texto, arg1, arg2);
696              for i in arg1..arg2 loop
697                   BorrarLinea(texto, i);
698              end loop;
699         end;

701         procedure CmdInsertarTexto(texto: in out TipoTexto;
702                               arg1: in out Natural; arg2: in out Natural) is
703              aux: TipoTexto;
704         begin
705              if arg1 /= NoArg and arg2 /= NoArg then
706                   CmdBorrarTexto(texto, arg1, arg2);
707              else
708                   if arg1 = NoArg then
709                        arg1 := texto.numlineas;
710                   end if;
711              end if;
712              LeerTexto(Standard_Input, aux, ".");
713              InsertarLineas(texto, arg1, aux);    -- texto se queda con aux
714         end;
```

```
716         procedure CambiarPalabra(linea: in out TipoLinea;
717                                  arg1: TipoString; arg2: TipoString) is
718              lista: TipoListaPalabra;
719         begin
720              lista := linea.primera;
721              while lista /= null  loop
722                   if IgualString(lista.palabra, arg1) then
723                        CopiarString(lista.palabra, arg2);
724                   end if;
725                   lista := lista.siguiente;
726              end loop;
727         end;

729         procedure CmdCambiarTexto(texto: in out TipoTexto;
730                                   arg1: TipoString; arg2: TipoString) is
731              lista: TipoListaLinea;
732         begin
733              lista := texto.primera;
734              while lista /= null  loop
735                   CambiarPalabra(lista.linea, arg1, arg2);
736                   lista := lista.siguiente;
737              end loop;
738         end;

740         procedure CmdGuardar(pedicion: in out TipoListaEdicion) is
741         begin
742              if pedicion.edicion.estado = Sucio then
743                   EscribirTexto(pedicion.edicion.texto);
744                   pedicion.edicion.estado := Limpio;
745              end if;
746         end;

748         procedure CmdCual(pedicion: in out TipoListaEdicion; imprestado: Boolean) is
749         begin
750              Put(AdaString(pedicion.edicion.nombre));
751              if imprestado and pedicion.edicion.estado = Sucio then
752                   Put(" sucio");
753              end if;
754              New_Line;
755         end;

757         procedure CmdListar(ediciones: in out TipoListaEdicion) is
758              lista: TipoListaEdicion;
759         begin
760              lista := ediciones;
761              while lista /= null loop
762                   CmdCual(lista, True);
763                   lista := lista.siguiente;
764              end loop;
765         end;
```

```
767         procedure CmdEditar(ediciones: in out TipoListaEdicion;
768                             pedicion: in out TipoListaEdicion;
769                             arg: String) is
770               lista: TipoListaEdicion;
771         begin
772               lista := ediciones;
773               pedicion := null;
774               while lista /= null and pedicion = null loop
775                     if AdaString(lista.edicion.nombre) = arg then
776                           pedicion := lista;
777                     else
778                           lista := lista.siguiente;
779                     end if;
780               end loop;
781               if pedicion = null then
782                     pedicion := new TipoNodoEdicion;
783                     NuevaEdicion(arg, pedicion.edicion);
784                     pedicion.siguiente := ediciones;
785                     ediciones := pedicion;
786               end if;
787         end;

789         procedure ProcesarComando( ediciones: in out TipoListaEdicion;
790                               pedicion: in out TipoListaEdicion;
791                               c: in out TipoComando) is
792         begin
793               case c.codigo is
794               when Insertar =>
795                     CmdInsertarTexto(pedicion.edicion.texto, c.args(1), c.args(2));
796                     pedicion.edicion.estado := Sucio;
797               when Borrar =>
798                     CmdBorrarTexto(pedicion.edicion.texto, c.args(1), c.args(2));
799                     pedicion.edicion.estado := Sucio;
800               when Imprimir =>
801                     CmdImprimirTexto(pedicion.edicion.texto, c.args(1), c.args(2));
802               when Cambiar =>
803                     CmdCambiarTexto(pedicion.edicion.texto, c.sargs(1), c.sargs(2));
804                     pedicion.edicion.estado := Sucio;
805               when Editar =>
806                     CmdEditar(ediciones, pedicion, AdaString(c.sarg));
807               when Guardar =>
808                     CmdGuardar(pedicion);
809               when Cual =>
810                     CmdCual(pedicion, False);  -- False: no imprimir estado
811               when Listar =>
812                     CmdListar(ediciones);
813               when Error =>
814                     null;
815               end case;
816         end;
```

```
818         ediciones: TipoListaEdicion;
819         pedicion: TipoListaEdicion;
820         comando: TipoComando;
821    begin
822         CmdEditar(ediciones, pedicion, "datos.txt");
823         while not End_Of_File loop
824              LeerComando(comando);
825              ProcesarComando(ediciones, pedicion, comando);
826              DestruirComando(comando);
827         end loop;
828         EscribirTexto(pedicion.edicion.texto);
829    end;
—
```

Esto que sigue es una sesión con el editor.

```
; ada e.adb
; e
datos.txt:  25 lineas
p 1 5
--
-- Escribe el contenido de datos.txt en su salida.
--
with Ada.Text_IO;
use Ada.Text_IO;
d 2
f
datos.txt
w
e e.adb
e.adb:  826 lineas
x
e.adb
datos.txt
```

¿Y ya está? Desde luego que no. Lo primero sería probrar *de verdad* el editor. Seguramente queden errores o, como suele llamárseles, *bugs* dentro del código. Aunque nosotros lo hemos probado y parece funcionar bien, habría que probarlo más, intentando romperlo. Es casi seguro que se puede.

Una vez roto hay que pensar qué hemos hecho para romperlo y arreglarlo. Sólo si el código y los datos están limpios resultará fácil arreglar los problemas.

Pero ahora es tiempo de tomarse un café, antes de seguir.

Problemas

1 Busca un buen libro de algoritmos y estructuras de datos y estúdialo. Podrías mirar el Wirth [1] si buscas un libro pequeño o el Cormen [2]. Pero cuidado, el Cormen se llama ‘‘Introduction to...’’ y como todos los libros que se titulan así este requiere algo de tiempo y paracetamol.

2 Lee el libro de Rob Pike y Kernighan sobre la práctica de la programación [3].

4 Lee cuanto código puedas, pero sólo si está escrito por personas que programen bien. No sólo se aprende por imitación a hacer obras de arte, también se aprende por imitación a hacer atrocidades y otros espantos. Por ejemplo, aunque la mayoría del código está en el lenguaje de programación C todo el código de Plan 9 from Bell Labs (`http://plan9.bell-labs.com`) está escrito por gente que programa muy bien.

5 Programa cuanto puedas.

Bibliografía

1. N. Wirth, *Algoritmos + Estructuras = Programas*, Editorial Castillo, 1999.
2. Cormen, *Introduction to Algorithms*, MIT Press.
3. B. W. Kernighan and R. Pike, *The Practice of Programming*, Addison-Wesley, 1999.

Indice Analítico

V

W

Post-Script

Este libro se ha formateado utilizando el siguiente comando

```
@{
        rfork n ; spanish
        pic title.ms | troff -ms
        eval ‘{doctype preface.ms}
        troff -ms toc.ms
        troff -ms prgtoc.ms
        labels -e  $CHAPSRC | bib -testd | pic| tbl | eqn | slant |  troff -ms
        troff -ms index.ms
        eval ‘{doctype epilog.ms}
} | lp -d stdout > fdp.ps
```

Muchas de las herramientas proceden de Plan 9, otras se han adaptado para el presente libro y otras se han programado expresamente para el mismo. Han sido precisos diversos lenguajes de programación incluyendo C, Rc, AWK y Troff para producir el resultado que ha podido verse impreso.