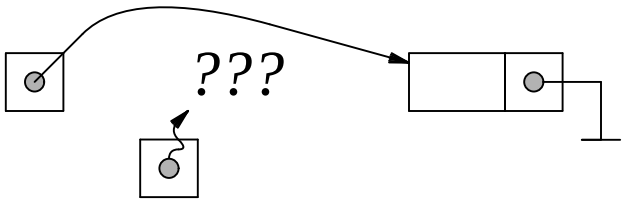
Curso Práctico de Programación



Usando Picky como Primer Lenguaje

Francisco J Ballesteros

Enrique Soriano

Gorka Guardiola

Prefacio

El presente libro es una reedición del ‘‘curso práctico de programación usando Ada como primer lenguaje’’, cuyo prefacio sigue a este. La diferencia fundamental es que en este utilizamos Picky como lenguaje de programación en lugar de Ada.

Los convenios del libro son los mismos que en la edición anterior, según puede verse en el prefacio de la edición anterior.

Picky es un lenguaje implementado por uno de los autores, Francisco J. Ballesteros, precisamente para la asignatura de Fundamentos de Programación, para solucionar los problemas derivados de utilizar Ada como lenguaje en dicha asignatura. Dicho autor debe agradecer a José Manuel Burgos, a Enrique Soriano y a Gorka Guardiola las discusiones relativas al diseño de Picky. El lenguaje está implementado para el sistema operativo Plan 9 de Bell Labs prestando especial atención a la portabilidad. Hay que agradecer a Gorka Guardiola y Enrique Soriano su adaptación a Linux, Windows y MacOS.

Esperamos que el lenguaje resulte útil para aprender a programar. Para nosotros, al menos, la transición de Ada a Picky en la preparación del material de la asignatura ha resultado como un soplo de aire fresco.

Francisco J. Ballesteros, Enrique Soriano, Gorka Guardiola.
{nemo, esoriano, paurea}@lsub.org

Laboratorio de Sistemas,
Universidad Rey Juan Carlos de Madrid
Madrid, España
2011

Prefacio a la primera edición

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 Pascal. 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`.

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 además el contenido de la memoria se pierde cuando apagamos el ordenador, se usa un disco en el que se 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 cómo escribir en la memoria los programas que queremos ejecutar y los datos que necesitan. 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).

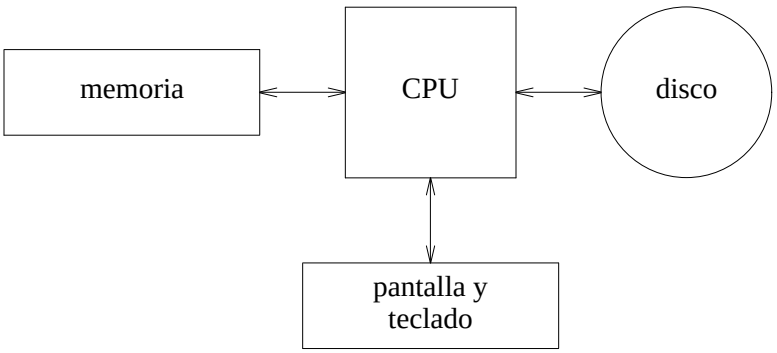


Figura 1.1: *Esquema de un ordenador.*

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 del 0 a 255 (ten en cuenta que si empezamos a asignar números desde el 0, al último elemento de los 256 elementos le corresponderá el número 255). 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 signifique 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 le denomina **software** (literalmente *parte blanda*, que se refiere a los componentes no tangibles). Al conjunto físico de cables, plástico, metal y circuitos se le denomina **hardware** (literalmente *parte dura* o tangible). Programar el ordenador consiste en introducirle un programa que efectúe las acciones que deseamos para resolver un problema. Dicho de otra forma, programar es construir software.

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

Antes de ver cómo programar, necesitamos conocer los 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 **Kilobyte** o **Kb**. Se supone que en realidad deberíamos decir *Kibibyte* y escribirlo *KiB*, pero en la práctica se sigue usando el término Kilobyte, aunque cada vez se usa más el símbolo *KiB*. 1024 Kilobytes hacen un **Megabyte** o **MB**. Igual que antes, debería decirse *Mebibyte*, cuyo símbolo es *MiB*, pero en la práctica se sigue usando el término Megabyte (aunque no es raro ver el símbolo *MiB*). 1024 Mbytes constituyen un **Gigabyte** o **GB** (el término preciso es *Gibibyte* y su símbolo es *GiB*). 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 un plan para programar. Lamentablemente, el hardware es tan tonto que no sabría qué hacer con un plan expresado en lenguaje natural (español). Pero a nosotros este plan nos puede servir perfectamente. Los pasos necesarios para 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 que 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 entienda 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, podemos ver si realmente está resolviendo el problema (y si lo está haciendo de la forma que queríamos al implementar el plan).

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 tengamos que probar 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 están aislados. No se hace uno por completo y luego se pasa al siguiente. Si queremos tener éxito, tenemos 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 general, el programa se acaba de escribir cuando el prototipo cumple con toda la especificación y se dice basta en este proceso de mejora a fuerza de repetir 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 ese 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 una secuencia detallada de acciones que define 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 una 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()
{
       tomar(sarten);
       if (cantidad(aceite) < Minima) {
              tomar(aceite);
              poneren(sarten, aceite);
       }
       poneren(fogon, sarten);
       encender(fogon);
       while (not estacaliente(aceite)) {
              nohacernada();
       }
       tomar(huevo);
       partir(huevo, cascara, interior);
       poneren(sarten, interior);
       eliminar(cascara);
       while (estacrudo(huevo)) {
              mover(sarten);/* para que no se pegue */
       }
       tomar(plato);
       quitarde(sarten, interior);
       poneren(plato, interior);
       apagar(fogon);
}
```

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 le denomina **sentencia**. Por ejemplo,

```
eliminar(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*, *Gedit*, o *SciTE*) y no un procesador de textos (un programa que sirve para escribir documentos, como *OpenOffice* o *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 código máquina, puesto que el programa que hace la traducción no lo entenderá. El texto del programa debe ser lo que se denomina *texto plano*.

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

Una vez codificado, tenemos que traducir el texto a código máquina. 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 el código fuente. Por tanto, **compilar** es traducir el programa al lenguaje del ordenador empleando otro programa para ello (este último ya disponible en el lenguaje del ordenador, claro).

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, el 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 se manda traducir por separado a chino mandarín (obteniendo el objeto del sofrito, del asado, etc.). Luego hay que reunir todos los objetos en una única receta en chino mandarín y dársela al pinche.

Una vez tenemos nuestros ficheros objeto, y los que hemos tomado 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. A estos ficheros se les llama **librerías** (o bibliotecas) del sistema. Las librerías son ficheros que contienen código ya compilado y listo para pasar al enlazador. Todo este proceso está representado esquemáticamente en la figura 1.2.

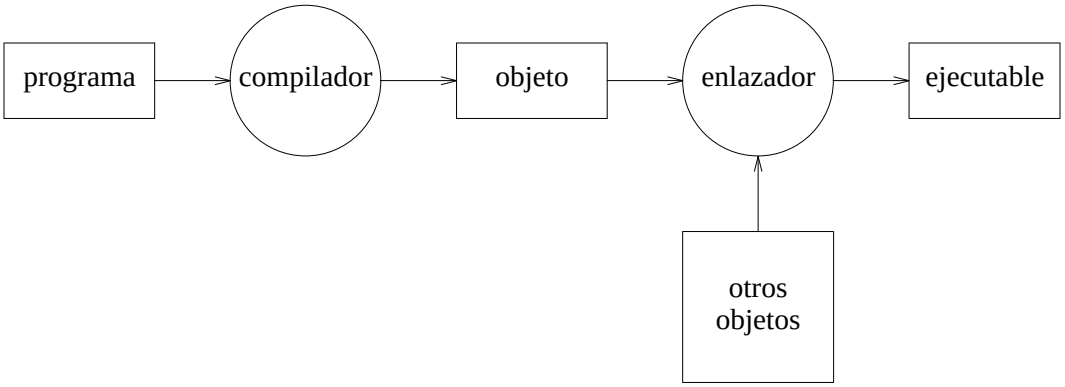


Figura 1.2: *El compilador genera ficheros objeto que se enlazan y forman un ejecutable.*

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 números de una longitud determinada que corresponden a órdenes concretas para la CPU. A estos números también se les conoce como **palabras**. El ejemplo anterior muestra cuatro palabras en cada línea. 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 o resultados (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.

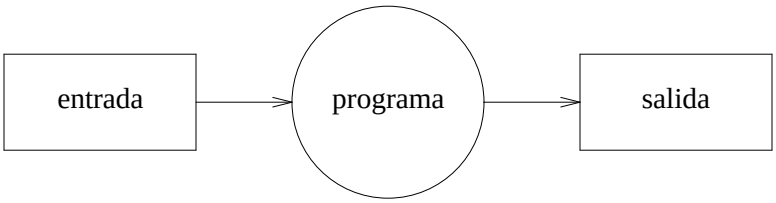


Figura 1.3: *Un programa toma datos de la entrada y tras procesarlos produce resultados en la salida.*

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 que puede detectar el compilador. El compilador también hace todo lo posible para cubrirnos las espaldas y detectar errores en el código que nos llevarían a una ejecución incorrecta. 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. Pero ten esto en cuenta: un programa puede compilar y no ser correcto. El compilador hace todo lo posible, pero no puede hacer magia.

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 el desastre 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.

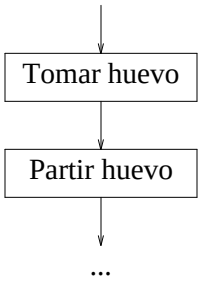


Figura 1.4: *Una secuencia ejecuta acciones una tras otra hasta concluir.*

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

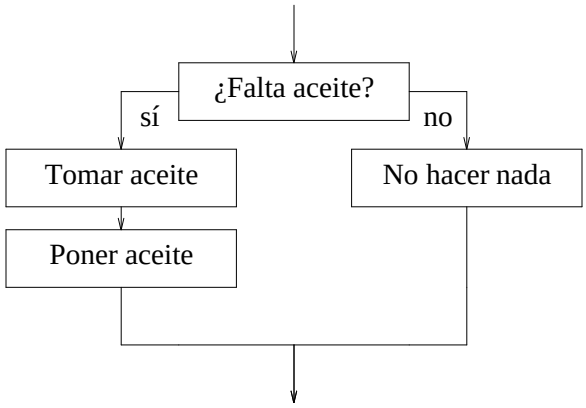


Figura 1.5: *Una selección ejecuta una u otra acción según una condición sea cierta o falsa.*

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.

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.

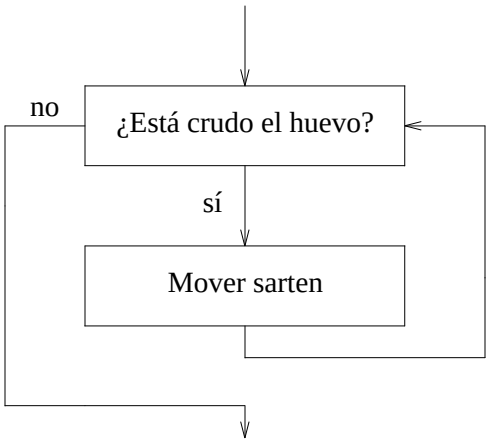


Figura 1.6: *Una iteración repite algunas sentencias mientras sea preciso.*

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 con un dedo índice 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 ni dedos en el ordenador. En su lugar, la CPU (la ‘‘mano’’) mantiene la cuenta de qué instrucción es la siguiente a ejecutar mediante el llamado **contador de programa** (el ‘‘dedo’’). 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 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 Picky

En este curso de programación vamos a usar un lenguaje creado específicamente para él, llamado Picky. Picky es un lenguaje de programación que puede utilizarse para escribir programas de ordenador, los cuales sirven por supuesto para manipular datos en el ordenador.

Picky, 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 Picky el nivel de abstracción del lenguaje está lejos del nivel empleado por el ordenador. Por eso decimos que Picky es un **lenguaje de alto nivel** (de abstracción, se entiende).

Esto es fácil de entender. Picky habla de conceptos (abstracciones) y el ordenador sólo entiende de operaciones elementales y números (elementos concretos y no muy abstractos). Por ejemplo, Picky 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 de este nivel de abstracción.

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 escrito en Picky.

Un programa Picky 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.p

```
1     /*
2      *    Programa de bienvenida en Picky.
3      *    Autor: aturing
4      */

6     /*
7      *    Nombre del programa.
8      */
9     program programa;

11    /*
12     *    Tipos de datos
13     */
14    types:
15         /* Dias de la semana */
16         TipoDiaSem = (Lun, Mar, Mier, Jue, Vie, Sab, Dom);
17
18    /*
19     *    Constantes
20     */
21    consts:
22         Pi = 3.1415926;
23         RadioPrueba = 2.0;

25    /*
26     *    Funciones y procedimientos.
27     */

29    function longitudcirculo(r: float): float
30    {
31         return 2.0 * Pi * r;
32    }

34    /*
35     *    Programa Principal.
36     */
37    procedure main()
38    {
39         writeln("Hola que tal.");
40         writeln(longitudcirculo(RadioPrueba));
41    }
```

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.p`.

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 Picky, 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 al compilador de Picky 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 están englobadas entre los símbolos ‘‘`/*`’’ y ‘‘`*/`’’. Estas líneas son **comentarios** y son líneas que el compilador ignora por completo. Esto es, todo el texto desde un ‘‘`/*`’’ hasta el ‘‘`*/`’’ (estos incluidos) es texto que podríamos borrar del programa si quisiéramos, y el programa seguiría siendo el mismo. No es necesario que esté en distintas líneas. Por ejemplo:

```
/* esto es un comentario */
```

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 fácilmente **legible** 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, las primeas líneas del programa anterior son un comentario:

```
1    /*
2     *    Programa de bienvenida en Picky.
3     *    Autor: aturing
4     */
```

Este comentario ayuda a que veamos una descripción del programa. A lo mejor, con leer ese comentario nos basta para ver qué hace, y nos ahorramos leer el programa entero.

Esta es la sintaxis que Picky 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 de ejemplo en Picky.
 *     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 se especifica el nombre del programa.

```
7    /*
8     *    Nombre del programa.
9     */
10   program programa;
```

Aquí, la palabra *program* es una palabra reservada que se utiliza para indicar que *programa* es a partir de este momento el nombre (o identificador) del programa que estamos escribiendo. Dado que el propósito de este programa es mostrar un programa en Picky, *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*. En ese caso, 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 de Picky ha de terminar en ‘‘`.p`’’. Por ejemplo, nuestro programa de ejemplo para esta disección debería estar en un fichero llamado ‘‘`programa.p`’’.

Como hemos visto, a lo largo del programa emplearemos palabras que tienen un significado especial. Aquí, la palabra *program* es una de esas palabras; decimos que es una **palabra reservada** o *palabra clave* del lenguaje (en inglés, *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 *program* para ningún otro propósito que para especificar el nombre del programa. Lo mismo pasa con las demás palabras reservadas.

En un programa también utilizamos otras palabras para referirnos a elementos en el lenguaje o a elementos en el mundo (según lo ve el programa, claro). Las llamamos identificadores. Por ejemplo, *programa* es un **identificador**. Los identificadores se llaman así porque identifican elementos en nuestro programa. En general, tendremos algunos identificadores ya definidos que podremos utilizar y también podremos definir otros nosotros. En este caso, como es el nombre de nuestro programa, lo hemos puesto 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 ayudásemos empleando una sintaxis y normas férreas, no sería posible implementar un compilador para el lenguaje.

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 Picky:

```
Imprimir_Linea
Put0
getThisOrThat
X32__
```

Pero estos otros **no** lo son:

```
0x10
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 segundo identificador de este ejemplo no es válido puesto que utiliza caracteres blancos (esto es, espacios en blanco) como parte del nombre.

Por cierto, Picky distingue entre mayúsculas y minúsculas. Los ordenadores, en general, hacen esta diferenciación. Por lo tanto, *Programa*, *programa*, *PROGRAMA* y *ProGraMa* son diferentes identificadores. Deberás recordar que esto es así en Picky, pero hemos de decir que en otros lenguajes no se distingue entre mayúsculas y minúsculas, y todos los ejemplos anteriores serían el mismo identificador.

Tras el nombre del programa, tenemos otra sección dedicada a definir nuevos elementos en el mundo donde vive nuestro programa. Esta sección se denomina sección de definición de tipos de datos (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.

```
13    /*
14     *    Tipos de datos
15     */
16    types:
17         /* Dias de la semana */
18         TipoDiaSem = (Lun, Mar, Mier, Jue, Vie, Sab, Dom);
```

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 esta definición, no es necesario usar un comentario. Con ver las siguientes dos líneas, se puede entender perfectamente que se trata de los días de la semana:

```
types:
       TipoDiaSem = (Lun, Mar, Mier, Jue, Vie, Sab, Dom);
```

En general, el código es más fácil de leer si no tenemos comentarios inútiles. Pero atención, esto no quiere decir que no tenga que haber comentarios. Esto quiere decir que hay que prescindir de los comentarios que no dan información útil, y que el código debe ser lo más autoexplicativo posible. A medida que avancemos con el curso, iremos viendo más ejemplos.

La palabra *types* es una palabra reservada del lenguaje que se utiliza (seguida por dos puntos) para marcar el inicio de la sección de definición de tipos de datos, y no podemos emplear ninguna otra en su lugar. En cambio, *TipoDiaSem*, *Lun*, *Mar*, etc. son identificadores, y podríamos haber empleado otros, ya que los identificadores los elige el programador.

La siguiente sección es la de definición de constantes. Como indica su nombre, son valores que nunca cambian (ya veremos para qué sirven en los siguientes capítulos). En este caso se usa la palabra reservada *consts* seguida de dos puntos para indicar el comienzo de la sección.

```
20    /*
21     *    Constantes
22     */
23    consts:
24         Pi = 3.1415926;
25         RadioPrueba = 2.0;
```

Aquí, de nuevo, vemos algunos identificadores como *Pi* y *RadioPrueba*, cuyos nombres dan una noción bastante clara de su propósito.

En estas líneas, podemos ver además que en un programa hay símbolos que se representan a sí 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, literalmente.

Como ya se ha comentado anteriormente, Picky distingue mayúsculas de minúsculas. 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 qué tipo de objeto se trata.

Las palabras reservadas se tienen que escribir siempre en minúscula. Los nombres de constantes los escribiremos siempre comenzando por mayúscula (con el resto en minúsculas). Ten en cuenta que al elegir nosotros mismos su identificador, podríamos escribirlo en minúsculas y para el compilador no habría ningún problema. Pero las vamos a escribir comenzando en mayúsculas para que, de un solo vistazo, podamos diferenciarlas de una palabra reservada. Así pues, *Pi* parece ser una constante. Los nombres de tipos los capitalizamos igual que los de constantes. Por ejemplo, *TipoDiaSem* en nuestro caso. Eso sí, 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 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:

```
27    /*
28     *    Funciones y procedimientos.
29     */

31    function longitudcirculo(r: float): float
32    {
33         return 2.0 * Pi * r;
34    }
```

Este fragmento de programa define la función *longitudcirculo*, similar a la función matemática empleada para la calcular la longitud 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 resulte innecesario. Así, una función llamada *longitudcirculo* seguramente no haga otra cosa que calcular la longitud de un círculo (de hecho, ¡no debería hacer otra cosa!).

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 *longitudcirculo(r)* sería el valor de $2\pi r$. Luego volveremos sobre esto, aunque puede verse que

```
33              return 2.0 * Pi * r;
```

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

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.

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

```
36    /*
37     *    Programa Principal.
38     */

40    procedure main()
41    {
42         writeln("Hola que tal.");
43         writeln(longitudcirculo(RadioPrueba));
44    }
```

Lo que hay entre ‘‘{’’ y ‘‘}’’ se conoce como el **cuerpo** del programa. A este programa lo llamamos **programa principal**, en consideración a que los subprogramas (funciones y procedimientos) que forman parte del fuente son en tambien programas. En cualquier caso, cuando ejecutemos el programa, éste empezará a realizar las acciones o sentencias indicadas en el cuerpo del programa principal. Eso sí, tras efectuar las definiciones encontradas antes.

En Picky, el programa principal siempre se llama *main*. Todo programa escrito en Picky tiene que tener un procedimiento llamado *main*.

Podemos ver que normalmente el programa principal 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. Como ya hemos visto antes, se usarán secuencias, selecciones e iteraciones para implementar el algoritmo.

En este programa principal utilizamos una sentencia que resultará muy útil a lo largo de todo el curso. La sentencia *writeln* escribe una línea en la pantalla con algo que le indicamos entre paréntesis. Hay otra sentencia, *write*, que también escribe, pero sin saltar a la siguiente línea. En

este ejemplo, el programa principal es una secuencia con dos sentencias: la primera escribe una línea en la pantalla saludando al usuario y la segunda escribe una línea en la pantalla con el resultado de calcular la longitud de un círculo.

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 llaves:

```
41    {
42          writeln("Hola que tal.");
43          writeln(longitudcirculo(RadioPrueba));
44    }
```

Por eso las sentencias en las líneas 42 y 43 están tabuladas a la derecha de ‘‘`{`’’ y ‘‘`}`’’ (debido a que dichas sentencias están dentro de ‘‘`{`’’ y ‘‘`}`’’ o dentro del cuerpo del programa). 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).

Todas las definiciones o declaraciones están sangradas o tabuladas; lo mismo que sucede con el cuerpo del programa (las sentencias entre las llaves).

Un último recordatorio: los signos de puntuación son importantes en un programa. 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 Picky, todas las sentencias terminan con un ‘‘`;`’’. 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, *procedure* no tiene nunca un punto y coma detrás.

2 Los signos tales como paréntesis y comillas deben de ir por parejas y bien agrupados. Esta sentencia

```
writeln("Hola que tal.")
```

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

```
writeln "Hola que tal.")
```

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 programa en Picky denominado ‘‘Hola π’’. Este programa resultará útil para los problemas que siguen y para el próximo capítulo.

holapi.p

```
1     /*
2      *    Saludar al numero π.
3      */

5     program holapi;

7     consts:
8          Pi = 3.1415926;

10    procedure main()
11    {
12         write("Hola ");
13         write(Pi);
14         write("!");
15         writeeol();
16    }
```

Este programa define una constante llamada *Pi* en la línea 8 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 ‘‘`pick`’’ al sistema operativo indicando el nombre del fichero que contiene el código fuente. Cuando damos una orden como esta al sistema operativo, estamos ejecutando un comando. En este caso, ‘‘ejecutamos el comando *pick*’’. Si el compilador no escribe ningún mensaje de error, entonces habrá sido capaz de traducir el código fuente y generar el programa. El nombre del fichero generado es, por omisión, `out.pam`. Para ejecutar el programa, basta con ejecutar el fichero binario generado. Por ejemplo:

```
; pick holapi.p
; out.pam
Hola 3.141593!
```

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

```
Hola 3.141593!
```

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

Puedes conseguir el software necesario para programar en Picky (el compilador) en

```
http://lsub.org/ls/picky.html
```

Problemas

1 Compila y ejecuta el programa ‘‘hola π’’ cuyo único propósito es escribir un saludo. Copia el código del programa 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 la línea que contiene la palabra reservada *program*. Compílalo y explica lo que pasa. Soluciona el problema.

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

7 Cambia el programa para que salude al mundo entero y no sólo al número π, por ejemplo, imprimiendo `hola mundo!`.

2 — Elementos básicos

2.1. ¿Por dónde empezamos?

En el capítulo anterior hemos tomado contacto con la programación y con el lenguaje Picky. 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 eso ayuda bastante). Si no tienes el compilador de Picky cerca, descárgalo de

```
http://lsub.org/ls/picky.html
```

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

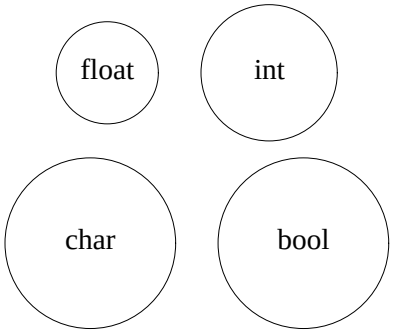


Figura 2.1: *El mundo según Picky: existen enteros, caracteres, reales y valores de verdad.*

¿Cómo es el mundo para Picky? En realidad es tan sencillo como muestra la figura 2.1. Para Picky, 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 Picky 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 le denomina **tipo de datos** y, naturalmente, a los elementos de cada conjunto se les 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 Picky 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.

Picky utiliza la palabra reservada *int* para denominar al tipo de datos (conjunto) de los números enteros. El conjunto de los reales está representado en Picky por el tipo de datos *float*. De igual manera, tenemos el conjunto de los valores de verdad, ‘‘cierto’’ y ‘‘falso’’, representado en Picky por el tipo de datos *bool*. Este último sólo tiene dos elementos: *True* representa en Picky 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 *char*.

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

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

Un entero sólo puede sumarse con otro entero; nunca con un número real. El punto entre el tres y el cero del segundo operando hace que ese literal sea 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:

```
; pick malo.p
malo.p:13: incompatible argument types (int and float) for op '+'
;
```

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 la siguiente sentencia (hemos cambiado la línea 13 de `holapi.p` para que imprima la suma del número entero con el número real):

```
writeln(2 + 3.0);
```

y esta sentencia no compila, dado que intenta mezclar 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 *int* 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 $3.2 \bullet 10^{-3}$, 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 real, igual que no tendría sentido convertir una pera en una manzana o el plomo en oro (bueno, esto último tal vez sí).

2.3. Operaciones

Como estamos viendo, cada tipo de datos tiene un nombre, por ejemplo *int*, 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 Picky de estas operaciones básicas (que son en realidad operaciones matemáticas) son los llamados **operadores**. Por ejemplo, ‘‘`+`’’ es un operador para el tipo *int* y podemos utilizarlo para sumar números enteros.

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)
- Conversión a otro tipo numérico: `float(3)`, `int(3.5)`
- Cambio de signo: `- 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 *int* dispone además del siguiente operador:

- Módulo: `5 % 3`

El módulo corresponde con el resto de la división entera. Por tanto, sólo está disponible para el tipo de datos *int*.

La lista de operadores la hemos incluido aquí más como una referencia que como otra cosa. Podemos olvidarnos de ella por ahora. 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 Picky el programa es simplemente una única secuencia de caracteres. Así es como ve Picky el programa para saludar la número π:

```
/*\n→enSaludar al numero π.\n */\n\nprogram holapi;\n\nconsts:
\n→enPi = 3.1415926;\n\nprocedure main()\n{\n→enwrite("Hola ");
\n→enwrite(Pi);\n→enwrite("!");\n→enwriteeol();\n}\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 *char* 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 *char*. 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. 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 Picky lo permite, no es así en otros lenguajes y no todo el software de ordenador (editores por ejemplo) se comporta igual con caracteres acentuados.

En ocasiones, es útil saber qué posición ocupa un carácter en la tabla ASCII. En Picky podemos obtener esta correspondencia mediante una conversión explícita del carácter a un entero. Por ejemplo, para saber la posición en la tabla del carácter `'Z'`, podemos usar la expresión:

```
int('Z')
```

Dicha expresión tiene como valor `90` dado que `'Z'` ocupa esa posición en el código. Para conseguir lo inverso, esto es, conseguir el carácter que corresponde a una posición en la tabla, podemos utilizar la conversión a *char*. Por ejemplo, esta expresión

```
char(90)
```

tiene como valor `'Z'`, del tipo *char*, dado que ese carácter tiene la posición 90 en el código de caracteres.

Otro tipo de datos importante es el llamado booleano, *bool* en Picky. 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`

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. Esto es:

```
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 son falsas!

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
```

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 odemos 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 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 entre sí:

```
<
>
<=
>=
==
!=
```

Podemos utilizarlos para comparar valores numéricos, caracteres y booleanos. Pero ambos operandos han de ser del mismo tipo. Estos operadores corresponden a las relaciones *menor que*, *mayor que*, *menor o igual que*, *mayor o igual que*, *igual a* y *distinto a*. `3 < 2` es *False*, pero `2 <= 2` es *True*.

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

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

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

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 ‘‘`!=`’’. Ya veremos más adelante cómo podemos comparar números reales.

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 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')
```

Ya comentamos antes que los caracteres son en realidad posiciones en una tabla. Las letras mayúscula están en dicha tabla de caracteres en posiciones consecutivas. Igual sucede con las minúsculas. Los dígitos también. Por tanto, podemos compararlos en base a su posición en la tabla. 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 Picky no podemos comparar

```
3 < '0'
```

dado que no hay **concordancia de tipos** entre `3` y `'0'`. El primero es de tipo entero, y el segundo de tipo carácter. 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 % 4) == 0 and ((a % 100) != 0 or (a % 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 % 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 % 4) == 0 and not ((a % 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 % 4) == 0 and (a % 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 % 100) != 0 or (a % 400) != 0
```

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

```
(a % 4) == 0 and ((a % 100) != 0 or (a % 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 Picky 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 Picky que implementa nuestro plan. Procediendo como hemos visto antes, llegamos a nuestra expresión

```
(a % 4) == 0 and ((a % 100) != 0 or (a % 400) == 0)
```

Por cierto, hemos usado paréntesis por claridad, pero podríamos haber escrito:

```
a%4 == 0 and (a%100 != 0 or a%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.p

```
1     /*
2      *    Es 2527 un año bisiesto?
3      */

5     program esbisiesto;

7     consts:
8          /*
9           *    A es el año que nos interesa
10          */
11         A = 2527;

13         /*
14          *    Esta constante booleana será cierta si A es bisiesto
15          */
16         EsABisiesto = (A % 4) == 0 and ((A % 100) != 0 or (A % 400) == 0);
17
18    /*
19     *    Programa Principal.
20     */

22    procedure main()
23    {
24         writeln(EsABisiesto);
25    }
—
```

Lo que resta es compilar y ejecutar el programa:

```
; pick esbisiesto.p
; out.pam
False
```

Cuando se ha compilado el programa, el complilador ha calculado el valor de las expresiones cuyos operandos son constantes (todas las de este programa tienen operandos constantes), y ha generado el fichero binario con las instrucciones del programa y los datos que necesita (incluyendo las constantes que hemos definido).

Cuando se ha ejecutado el binario, se ha empezado ejecutando su procedimiento principal. En este caso, el programa principal tiene una única sentencia. Dicha sentencia ha escrito el valor de nuestra constante *EsABisiesto* 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 se calcula en Picky el valor de las constantes? Se calcula el valor correspondiente a la expresión durante la compilación del programa. Calcular el valor de una expresión se denomina **evaluar** la expresión. La constante queda definida con el valor resultante tras evaluar la expresión. En el últimpo ejemplo, el compilador ha calculado un valor de tipo *bool* para la expresión a partir de la cual hemos definido la constante *EsABisiesto*. Luego la constante *EsABisiesto* será de tipo *bool*. Cuando el programa ejecute, siempre que use la constante definida, esta tendrá el mismo valor (el que se calculó cuando se compiló el programa).

Nótese que no todas las expresiones que aparecen en un programa se calculan al compilar el programa. Si la expresión tiene operandos que no son constantes, la expresión se evalúa al ejecutar el programa. En realidad, ese será el caso común, como veremos más adelante en el curso. En todo caso, la evaluación de una expresión se realiza de la misma forma en ambos casos.

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 % 4) == 0 and ((A % 100) != 0 or (A % 400) == 0)
```

y *A* tiene como valor `444`, entonces Picky evalúa

```
(444 % 4) = 0 and ((444 % 100) != 0 or ((444 % 400) == 0))
```

Aquí, Picky va a calcular primero `444 % 4` o bien `444 % 100` o bien `444 % 400`. Estas subexpresiones son las más internas o más anidadas de la expresión. Supongamos que tomamos la segunda. Esto nos deja:

```
(444 % 4) == 0 and (44 != 0 or ((444 % 400) == 0))
```

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

```
(444 % 4) == 0 and (44 != 0 or (44 == 0))
(444 % 4) == 0 and (44 != 0 or False)
(444 % 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). En Picky, los operadores con la misma precedencia se evalúan de izquierda a derecha (ten cuidado, esto no es así en todos los lenguajes de programación).

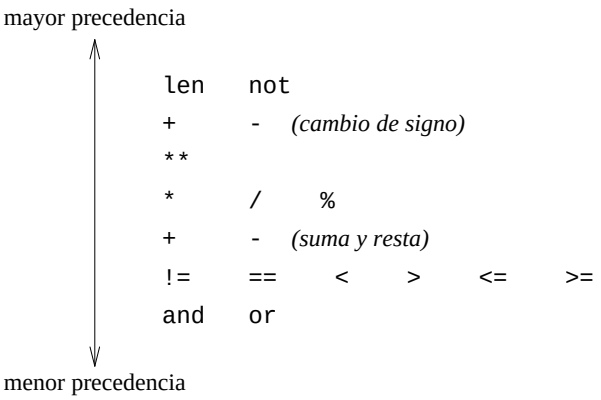


Figura 2.2: *Precedencia de operadores en Picky.*

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
```

Por cierto, hay dos leyes muy útiles en expresiones booleanas que son las denominadas **leyes de 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?

Un último detalle importante sobre las expresiones. Como ya se explicó en el capítulo anterior, el compilador hace todo lo posible por detectar fallos en el programa. Algunos errores en expresiones se detectan al compilar el programa, generando un error de compilación.

El compilador evalúa las expresiones cuyos operandos son constantes, ya que los valores de las constantes se conocen cuando el compilador traduce el programa. Gracias a esto, el compilador es capaz de detectar ciertos errores en las expresiones evitando que el programa tenga errores de ejecución. Por ejemplo, si en una expresión se está dividiendo entre una constante con valor cero o se está calculando la raíz cuadrada de una constante con un valor negativo, el compilador se negará a generar el fichero binario y dará errores de compilación. Pero atención, esto no nos libra de tener otros errores lógicos o de ejecución en nuestros programas.

2.8. Elementos predefinidos en Picky

En Picky existen funciones, operaciones y constantes predefinidas en el lenguaje (en inglés, *built-in*) que resultan muy útiles para construir expresiones. Estas funciones y constantes pueden ayudarnos a obtener valores que dependen de un tipo de datos determinado.

Los tipos ordinales, esto es, los tipos de datos cuyos valores se pueden contar y tienen un orden, tienen definidas las funciones

```
pred(v)
succ(v)
```

que, respectivamente, evalúan al valor predecesor (anterior) y sucesor (posterior) al valor *v* (en el tipo de datos correspondiente). Por ejemplo, si *c* es de tipo *char* con valor `'B'`, entonces

```
pred(c)
```

tiene como valor `'A'`. Siguiendo con el mismo ejemplo,

```
succ(c)
```

tiene como valor el siguiente valor en el tipo de datos *char*, que es `'C'`.

También tenemos funciones predefinidas para los números reales que son muy útiles para escribir expresiones, como la raíz cuadrada, el seno, el coseno, etc.

Built-in	Función
`acos(r)`	arcocoseno
`asin(r)`	arcoseno
`atan(r)`	arcotangente
`cos(r)`	coseno
`exp(r)`	exponencial
`log(r)`	logaritmo
`log10(r)`	logaritmo base 10
`pow(r1, r2)`	potencia
`sin(r)`	seno
`sqrt(r)`	raíz cuadrada
`tan(r)`	tangente

Figura 2.3: *Funciones predefinidas para números reales.*

Un *built-in* llamado *fatal* nos permite abortar la ejecución del programa en cualquier punto. Esto sólo se debe hacer cuando nuestro programa ha detectado una situación de la que no se puede recuperar. Ya veremos ejemplos durante el curso. Por ejemplo, si el programa llega a ejecutar la sentencia

```
fatal("Error muy grave!");
```

entonces se acaba su ejecución y se imprime por la pantalla ‘‘`fatal: Error muy grave!`’’. A medida que avance el libro, irán apareciendo nuevos *built-in*.

Picky también tiene constantes predefinidas. Por ejemplo, Picky tiene constantes para el mínimo y el máximo valor de los tipos *int* y *char*. Así podemos utilizar estos valores en las expresiones que construimos. *Minint* representa el mínimo valor entero y *Maxint* representa el máximo valor entero. Por ejemplo, estas dos sentencias escribirían por la pantalla el máximo entero y su predecesor:

```
writeln(Maxint);
writeln(pred(Maxint));
```

Igualmente, *Minchar* y *Maxchar* representan el mínimo y el máximo valor de un carácter.

No podemos usar los nombres de las funciones y las constantes predefinidas como identificadores en nuestros programas, ya que esos nombres están reservados para el propio lenguaje (como las palabras reservadas).

Los *built-ins* se pueden usar en cualquier expresión del programa, incluyendo las expresiones que escribimos para definir nuestras propias constantes. Por ejemplo:

```
consts:
      C = sqrt(120.0) + 22.0;
```

En este ejemplo, la constante real *C* tendrá el valor resultante de sumar 22.0 a la raíz cuadrada de 120.0.

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...

calculocircunferencia.p

```
1     /*
2      *    Programa para calcular la longitud de
3      *    la circunferencia de un circulo de radio r.
4      */

6     program calculocircunferencia;

8     /*
9      *    Programa Principal.
10     */
11    procedure main()
12    {
13         ;
14    }
```

No es que haga mucho. De hecho, hemos utilizado la sentencia nula como cuerpo del programa principal: dejando únicamente el punto y coma. Esto quiere decir que Picky 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. Evidentemente, no veremos nada en la pantalla al ejecutar el programa, porque no hace nada.

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.

calculocircunferencia.p

```
1     /*
2      *    Programa para calcular la longitud de
3      *    la circunferencia de un circulo de radio r.
4      */

6     program calculocircunferencia;

8     consts:
9          /*
10          *    Número Pi
11          */
12         Pi = 3.1415926;

14    /*
15     *    Programa Principal.
16     */
17    procedure main()
18    {
19         ;
20    }
```

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 definir otra constante para el radio que nos interesa para el cálculo, *Radio*, y otra más para la expresión que queremos calcular. Podríamos incluir esto justo debajo de la declaración para *Pi*:

```
16     Radio = 3.0;
17     LongitudCircunferencia = 2.0 * Pi * Radio;
```

Por último necesitamos escribir el resultado de nuestro cálculo. Para ello cambiamos la sentencia

nula por

```
writeln(LongitudCircunferencia);
```

en el cuerpo del programa. El programa resultante es como sigue:

calculocircunferencia.p

```
/*
 *   Programa para calcular la longitud de
 *   la circunferencia de un circulo de radio r.
 */

program calculocircunferencia;

consts:
     /*
      *   Número Pi
      */
     Pi = 3.1415926;
     /*
      *   Radio que nos interesa y long. circunf.
      */
     Radio = 3.0;
     LongitudCircunferencia = 2.0 * Pi * Radio;

/*
 *   Programa Principal.
 */

procedure main()
{
     writeln(LongitudCircunferencia);
}
```

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

```
; pick calculocircunferencia.p
; out.pam
18.849556
```

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:

```
/*
 *   Función que calcula la longitud de la
 *   circunferencia de radio r.
 */
function longitudcircunferencia(r: float): float
{
     return 2.0 * Pi * r;
}
```

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 ‘‘`{`’’ y ‘‘`}`’’ 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:

calculocircunferencia.p

```
1     /*
2      *    Programa para calcular la longitud de
3      *    la circunferencia de un circulo de radio r.
4      */

6     program calculocircunferencia;

8     consts:
9          /*
10          *    Número Pi.
11          */
12         Pi = 3.1415926;

14         /*
15          *    Radio que nos interesa.
16          */
17         Radio = 3.0;

19    /*
20     *    Función que calcula la longitud de la
21     *    circunferencia de radio r.
22     */
23    function longitudcircunferencia(r: float): float
24    {
25         return 2.0 * Pi * r;
26    }

28    /*
29     *    Programa principal.
30     */
31    procedure main()
32    {
33         write("Longitud circunferencia= ");
34         write(longitudcircunferencia(Radio));
35         writeeol();
36    }
```

—

En el cuerpo del programa escribimos un mensaje explicativo, después escribimos el valor de la expresión para calcular la longitud de la circunferencia y, por último, pasamos a una línea nueva en la pantalla (escribimos un fin de línea). Para escribir por pantalla la longitud se ha escrito una expresión que usa sólo la función que acabamos de definir:

```
write(longitudcircunferencia(Radio));
```

La expresión

```
longitudcircunferencia(Radio)
```

tiene como valor (calculará) la longitud de la circunferencia de radio *Radio* (*Radio* es la constante que tenemos definida con el radio que nos interesa, en este caso 3.0). Esta es la compilación y ejecución del programa:

```
; pick calculocircunferencia.p
; out.pam
Longitud circunferencia= 18.849556
```

En adelante utilizaremos funciones para realizar nuestros cálculos, como en este último programa.

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 Picky 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 predefinidas en Picky, tales como la raíz cuadrada. Consulta el libro para encontrar las funciones necesarias.

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 $b = 3$

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{a!}{b!(a-b)!}$$

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 Picky que imprima su valor.

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

b) Hazlo ahora 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): float
2 {
3     return a + b;
4 }
```

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. Por ejemplo, en el capítulo anterior utilizamos una función para calcular la longitud de una circunferencia. 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): float
```

Esta línea forma parte de la definición de la función Picky *longitudcircunferencia*, y se la denomina la **cabecera de función** para la función *longitudcircunferencia*. 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 (qué tipo de datos devuelve 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 Picky 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 Picky 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. En una cabecera de Picky, el tipo de dato devuelto se especifica poniendo dos puntos y el tipo, detrás de los parámetros:

```
function areacirculo(r: float): float
```

¡Esta línea es la definición en Picky 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).

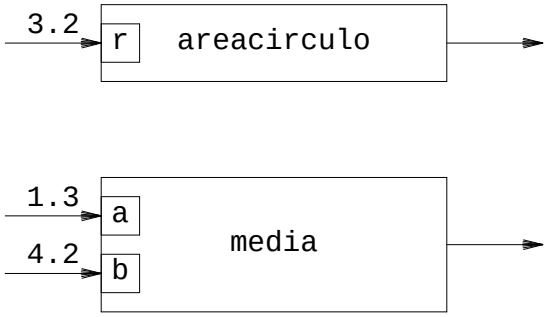


Figura 3.1: *Dos funciones en Picky. Tienen argumentos y devuelven resultados.*

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). Por ejemplo:

```
function media(num1: float, num2: float): float
```

Volviendo a nuestro problema de ejemplo, resta implementar la solución. La forma de hacer esto en Picky 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): float
2 {
3     return Pi * r ** 2.0;
4 }
```

La sentencia *return*, como sabemos, hace que la función devuelva un valor como resultado.

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

calculoarea.p

```
1     /*
2      *    Programa para calcular el area
3      *    de un circulo de radio r.
4      */

6     program calculoarea;

8     consts:
9          Pi = 3.1415926;
10         Radio1 = 3.2;   /* una prueba */
11         Radio2 = 4.0;   /* otra prueba */

13    function areacirculo(r: float): float
14    {
15         return Pi * r ** 2.0;
16    }

18    /*
19     *    Programa Principal.
20     */
21    procedure main()
22    {
23         writeln(areacirculo(Radio1));
24         writeln(areacirculo(Radio2));
25    }
—
```

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 compilar el programa, se definieron las constantes. Al ejecutar este programa, se comenzarán a ejecutar las sentencias del cuerpo del programa principal. Cuando llegue el momento de evaluar *areacirculo(Radio1)*, se ‘‘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. Después ejecutará la segunda sentencia del programa principal, que llama a la función usando *Radio2* como argumento (cuyo valor es 4.0). Cuando compilamos y ejecutamos el programa, podemos ver:

```
; pick calculoarea.p
; out.pam
32.169914
50.265488
```

Podemos utilizar llamadas a función en cualquier lugar donde podemos utilizar un valor del tipo que devuelve la función considerada (salvo para declarar constantes). 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 Picky 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 Picky deja lo que está haciendo y llama a *areacirculo* utilizando 9.0 como valor para el parámetro *r*. Eso hace que Picky calcule

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

lo que tiene como valor `254.469025`. La sentencia *return* de la función hace que se devuelva este valor. Ahora Picky continúa con lo que estaba haciendo, evaluando nuestra expresión:

```
(3.0 + 254.469025) / 2.0
257.469025 / 2.0
128.734512
```

Resumiendo todo esto, podemos decir que una función en Picky 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 Picky, 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 Picky encuentra *areacirculo(3.0)* en una expresión, 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 los capítulos anteriores. La declaración de constantes se realiza en una sección del programa especial, que comienza con la palabra reservada *consts* seguida de dos puntos, y todas las definiciones de constantes que necesitemos. Por ejemplo:

```
consts:
       Pi = 3.1415926;
       Maximo = 3;
       Inicial = 'B';
```

Para definir una constante en la sección, primero se escribe el identificador de la constante, *Pi*, seguido de ‘‘=’’ seguido del valor de la constante. Como de costumbre, también hay que escribir un ‘‘;’’ para terminar la declaración. La constante tendrá el tipo correspondiente al valor que se le ha asignado en su definición. En el ejemplo de más arriba, *Inicial* es una constante de tipo *char*, *Maximo* es una constante de tipo *int*, y *Pi* es una constante de tipo *float*.

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(): int
2  {
3      return 5 * 4 * 3 * 2 * 1;
4  };
```

Ha bastado seguir estos pasos:

1 Buscar la definición de factorial de un número.

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

3 Escribir la expresión en Picky 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.

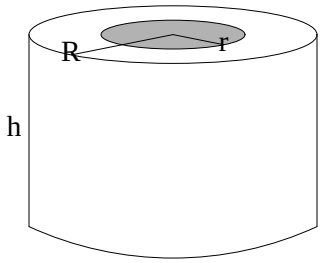


Figura 3.2: *Un sólido cilíndrico perforado por un cilindro.*

Lo primero es definir el problema. ¿Qué necesitamos saber para que esté definido nuestro sólido? Necesitamos la altura del cilindro, *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 volcilindrohueco(altura: float, rmin: float, rmax: float): 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  /*
2   *  Calcula el volumen de un cilindro de altura y radio (rmax)
3   *  conocidos taladrado por otro cilindro de radio dado (rmin)
4   */
5  function volcilindrohueco(altura: float,
6             rmin: float,
7             rmax: float): float
8  {
9      return ¿¿AreaBase?? * altura;
10 }
```

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 Picky como la cabecera de función:

```
/*
 *     Area de corona circular dados los radios interno y externo
 */
function areacorona(rmin: float, rmax: float): float
```

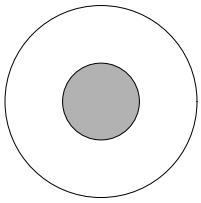


Figura 3.3: *Una corona circular (un círculo con un agujero circular en su centro).*

Por lo que en realidad nuestro programa debe ser:

```
1  /*
2   *  Calcula el volumen de un cilindro de altura y radio (rmax)
3   *  conocidos taladrado por otro cilindro de radio dado (rmin)
4   */
5  function volcilindrohueco(altura: float,
6             rmin: float,
7             rmax: float): float
8  {
9      return areacorona(rmin, rmax) * altura;
10 }
```

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  /*
2   *  Area de corona circular dados los radios interno y externo
3   */
4  function areacorona(rmin: float, rmax: float): float
5  {
6      return areacirculo(rmax) - areacirculo(rmin);
7  }
```

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

volcilindro.p

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

6     program volcilindro;

8     consts:
9          Pi = 3.1415926;
```

```
11    function areacirculo(r: float): float
12    {
13         return Pi * r ** 2.0;
14    }

16    /*
17     *   Area de la corona circular dados los radios interno y externo
18     */
19    function areacorona(rmin: float, rmax: float): float
20    {
21         return areacirculo(rmax) - areacirculo(rmin);
22    }

24    /*
25     *   Calcula el volumen de un cilindro de altura y radio (rmax)
26     *   conocidos taladrado por otro cilindro de radio dado (rmin)
27     */
28    function volcilindrohueco(altura: float,
29              rmin: float,
30              rmax: float): float
31    {
32         return areacorona(rmin, rmax) * altura;
33    }

35    /*
36     *   Programa Principal.
37     */
38    procedure main()
39    {
40         writeln(volcilindrohueco(3.0, 2.0, 3.0));
41    }
```

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 *volcilindrohueco*. ¡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).

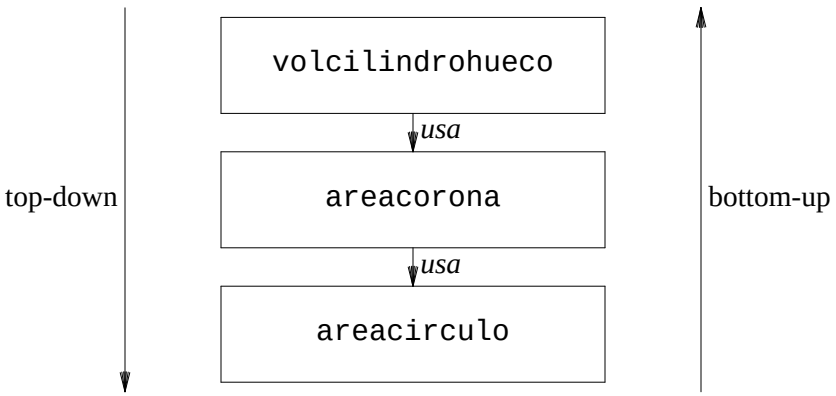


Figura 3.4: *Podemos programar de arriba-a-abajo o de abajo-a-arriba. En realidad, hay que usar ambas.*

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 Picky 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 *write* para escribir por pantalla 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.p

```
1     /*
2      *    Programa para escribir un numero de tres
3      *    dígitos espaciado
4      */

6     program digitosseparados;

8     consts:
9          Numero = 325;

11    /*
12     *    Devuelve el valor del digito en posicion dada.
13     *    La primera posicion es 1.
14     */
15    function valordigito(numero: int, posicion: int): int
16    {
17         return 0;
18    }

20    /*
21     *    Programa Principal.
22     */
23    procedure main()
24    {
25         write(valordigito(Numero, 3));
26         write("   ");
27         write(valordigito(Numero, 2));
28         write("   ");
29         write(valordigito(Numero, 1));
30         writeeol();
31    }
```

—

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

```
; pick digitosseparados.p
; out.pam
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 Picky 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: int, pos: int): int
2  {
3      return (numero / 10 ** (pos - 1)) % 10;
4  }
```

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.

```
; pick digitosseparados.p
; out.pam
       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:

```
15 function valordigito(numero: int, posicion: int): int
16 {
17     return (numero / Base ** (pos - 1)) % Base;
18 }
```

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:

```
14 /*
15  *  Devuelve el valor del digito en base y posicion dada.
16  *  La primera posicion es 1.
15   */
17 function valordigito(numero: int,
18                   pos:int,
19                   base:int): int
20 {
21     return (numero / base ** (pos - 1)) % base;
22 }
```

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. Para obtener la posición, convertiremos el *char* a *int*, como hemos visto anteriormente en este capítulo:

```
1  function valorcarnumerico(car: char): int
2  {
3      return int(car) - int('0');
4  }
```

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: int): char
2  {
3      return char(int('0') + digito);
4  }
```

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 Picky tenemos constantes predefinidas para estos caracteres *ASCII*: la constante *Tab* para el tabulador (horizontal), la constante *Eol* para el fin de línea. Como el problema consiste en ver si algo se cumple o no, deberíamos devolver un valor de verdad.

```
1  function esblanco(c: char): bool
2  {
3      return c == ' ' or c == Tab or c == Eol;
4  }
```

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 el módulo. 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 anteriormodn(num: int, n: int): int
2  {
3      return (num + n - 1) % n;
4  }
```

3.5.6. Comparar números reales

Ya se explicó anteriormente que los *float* no se deben comparar directamente con el operador de igualdad ‘‘==’’ ya que en realidad son aproximaciones al número real y, seguramente, no coincidan todos los decimales de los dos números. Por ejemplo, este programa:

```
1  program compararreales;
2
3  consts:
4      A =  2.0 - 1.1;
5      B =  1.0 - 0.1;
6
7  procedure main()
8  {
9      writeln(A == B);
10 }
```

Viendo las expresiones que dan valor a las constantes, diríamos que las constantes son iguales (0.9). Pero si compilamos y ejecutamos, veremos que no lo son, difieren el algún decimal:

```
; pick compararreales.p
; out.pam
False
```

En lugar de comparar los números reales con ‘‘==’’, es 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 (un valor arbitrariamente pequeño). Veremos como programarlo. Si damos por hecho que tenemos una forma de calcular el valor absoluto (*abs*) y una constante con el error que contemplamos para la comparación (*Epsilon*) la función de comparación sería la siguiente:

```
7  function sonrealesiguales(a: float, b: float): bool
8  {
9      return abs(a - b) < Epsilon;
10 }
```

Ahora nos quedaría resolver el problema de calcular el valor absoluto para completar el programa, que quedaría como sigue (ahora escribe por su salida la comparación mal hecha y la comparación bien hecha).

compararreales.p

```
1     /*
2      *    Programa para aprender a comparar numeros
3      *    reales.
4      */

6     program compararreales;

8     consts:
9          Epsilon = 0.0005;

11         /*
12          *    constantes para probar
13          */
14         A =  2.0 - 1.1;
15         B =  1.0 - 0.1;

17    function abs(a: float): float
18    {
19         return sqrt(a ** 2.0);
20    }

22    function sonrealesiguales(a: float, b: float): bool
23    {
24         return abs(a - b) < Epsilon;
25    }

27    procedure main()
28    {
29         write("Comparacion mal: ");
30         writeln(A == B);
31         write("Comparacion bien: ");
32         writeln(sonrealesiguales(A, B));
33    }
```

Si compilamos y ejecutamos el programa:

```
; pick compararreales.p
; out.pam
Comparacion mal: False
Comparacion bien: True
```

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.

Como has podido observar, para escribir por la salida estándar un dato se usa *write* o su variante *writeln*, que es igual que *write*, pero acaba la línea después de escribir el dato que le pasamos como argumento. Como nota diremos que, en realidad, *write* acepta argumentos de distintos tipos. En base al tipo del argumento, actuará de una forma u otra (no hará lo mismo para escribir un entero que para escribir un real). A esto se le llama **polimorfismo**. Picky lo usa para algunas de sus funciones predefinidas.

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.

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 Picky 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 Carácter que se corresponde a la posición 87 en la tabla ASCII.

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).

53 Ver si tres números son primos entre sí. Un número es primo con respecto a otro si no son divisibles entre sí.

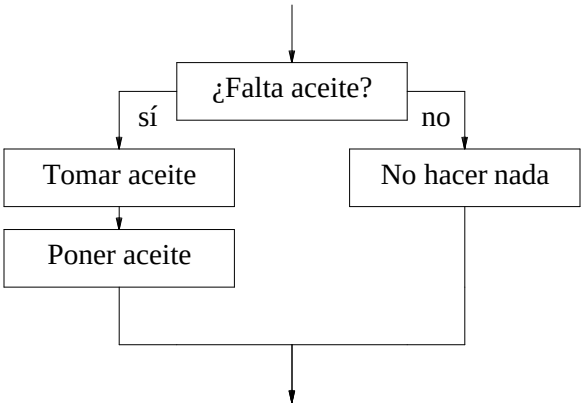
54 Ver cuantos semitonos hay entre dos notas musicales.

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 Picky, tal y como hemos visto en el capítulo anterior.

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 Picky 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){          /* si ... entonces */
        sentencias
}else{                  /*  si no */
        sentencias
}                       /*  fin si */
```

Cuando Picky tiene que ejecutarla procede como sigue:

1 Se evalúa la condición, que ha de ser siempre una expresión de tipo *bool*. Esto es, una expresión que corresponde a un valor de verdad. La expresión tiene que estar rodeada por paréntesis.

2 Si la condición se cumple (esto es, tiene como valor *True*) entonces se ejecutan las sentencias que hay entre las llaves a continuación del `if`. A ese conjunto de sentencias se le llama la ‘‘rama *then*’’). En otro caso se ejecutan las sentencias que se encuentran entre las llaves a continuación de la palabra reservada *else* (la ‘‘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: int, b: int): int
2  {
3      if(a > b){
4             return a;
5      }else{
6             return b;
7      }
8  }
```

En este punto merece la pena que vuelvas atrás y compares la definición de la función matemática definida a trozos con la función Picky 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 los paréntesis. 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* de la estructura de control *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){          /* si ... entonces */
        sentencias
}                       /* 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 Picky como sigue:

```
1  /*
2   *  -1, 0, 1 para un numbero negativo, cero o positivo
3   */
4  function signo(n: int): int
5  {
6      if(n > 0){
7             return 1;
8      }else{
9             if(n == 0){
10                   return 0;
11            }else{
12                   return -1;
13            }
14     }
15 }
```

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){          /* si ... entonces */
       sentencias
}else if(condición){    /* si no, si ... entonces */
       sentencias
}else if(condición){    /* si no, si ... entonces */
       sentencias
}else{                  /* si no */
       sentencias
}                       /*  fin si */
```

Esto es sólo ayuda sintáctica, para que no tengamos anidar if-then-else unos dentro de otros. Pero es cómodo. Esta es la función `signo` programada de un modo un poco mejor:

```
1  /*
2   *  -1, 0, 1 para un numbero negativo, cero o positivo
3   */
4  function signo(n: int): int
5  {
6      if(n > 0){
7             return 1;
8      }else if(n == 0){
9             return 0;
10     }else{
11            return -1;
12     }
13 }
```

Hasta ahora hemos utilizado los *if-then* para definir funciones en Picky. 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, en base a la edad de una persona, si es adulto o es menor de edad:

```
if(Edad < 18){
       writeln("eres menor de edad");
}else{
       writeln("eres adulto");
}
```

Hay una tercera estructura de control para ejecución condicional de sentencias. Esta estructura se denomina *case* (a veces se la llama *switch*) 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 *int* o de tipo *char*, por ejemplo). Su uso está indicado cuando hay múltiples casos excluyentes, de ahí su nombre. La estructura de la sentencia *case* (o *switch*) es como sigue:

```
switch(expresión){
case valores:
       sentencias
case valores:
       sentencias
case valores:
       sentencias
 ...
default:
       sentencias
}                   /* fin del switch */
```

Picky 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 un *case*, entonces se ejecuta la rama *default*. Por último, igual que de costumbre, la ejecución continúa con las sentencias escritas tras la estructura de control (tras el fin del *switch*).

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*. La hemos programado algo más complicada de lo que podría ser, para que veas varias formas de escribir valores tras un *case*.

```
1  /*
2   *  Ojo! esta funcion tiene un error! ¿lo ves?
3   */
4  function valordigito16(digito: char): int
5  {
6      switch(digito){
7      case '0':
8             return 0;
9      case '1':
10            return 1;
11     case '2', '3':
12            return int(digito) - int('0');
13     case 'A'..'F':
14            return int(digito) - int('A') + 10;
15     default:
16            return 0;
17     }
18 }
```

Cuando Picky 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                 case ’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                 case ’2’, ’3’:
9                         return int(digito) - int(’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                case ’A’..’F’:
11                        return int(digito) - int(’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 *default* en la mayoría de los casos.

¿Has encontrado el error en la función *valordigito16* anterior? Cuando el carácter *digito* es, por ejemplo, `’6’` ¡Se ejecuta la rama *default*! Por tanto, el valor devuelto por la función es 0, cuando debería devolver un 6. Esta función no hace bien su trabajo, tiene un error lógico. En realidad, tiene ese error cuando *digito* tiene algún valor comprendido entre los caracteres `’4’` y `’9’`.

La función se habría podido escribir de un modo más compacto utilizando:

```
1  function valordigito16(digito: char): int
2  {
3      switch(digito){
4      case ’0’..’9’:
5             return int(digito) - int(’0’);
6      case ’A’..’F’:
7             return int(digito) - int(’A’) + 10;
8      default:
9             return 0;
10     }
11 }
```

pero hemos querido aprovechar para mostrar las formas típicas de seleccionar los distintos casos. Esta nueva versión no tiene el error lógico de la anterior.

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  /*
2   *  Posicion del punto mas distante al origen en una recta.
3   */
4  function masdistante(a: int, b: int): int
5  {
6      if(distancialineal(Origen, a) > distancialineal(Origen, b)){
7             return a;
8      }else{
9             return b;
10     }
11 }
```

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  /*
2   *   Distancia entre pos puntos en una recta discreta.
3   */
4  function distancialineal(a: int, b: int): int
5  {
6      if(a > b){
7             return a - b;
8      }else{
9             return b - a;
10     }
11 }
```

Otra forma de hacerlo, una vez tenemos programada la función *signo* vista al principio del capítulo, es esta:

```
1  /*
2   *   Distancia entre pos puntos en una recta discreta.
3   */
4  function distancialineal(a: int, b: int): int
5  {
6      return (a - b) * signo(a - b);
7  }
```

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.p

```
1     /*
2      *    Programa para calcular distancias.
3      */

5     program distancia;
```

```
7     consts:
8          /*
9           *    Ponemos nuestro punto origen en el 2
10          */
11         Origen = 2;

13    /*
14     *    -1, 0, 1 para un numbero negativo, cero o positivo
15     */
16    function signo(n: int): int
17    {
18         if(n > 0){
19              return 1;
20         }else if(n == 0){
21              return 0;
22         }else{
23              return -1;
24         }
25    }

27    /*
28     *    Valor absoluto de un entero
29     */
30    function valabs(n: int): int
31    {
32         return n * signo(n);
33    }

35    /*
36     *    Distancia entre dos puntos en una recta
37     */
38    function distancialineal(a: int, b: int): int
39    {
40         return valabs(a - b);
41    }

43    /*
44     *    Posicion del punto mas distante al origen en una recta
45     */
46    function masdistante(a: int, b: int): int
47    {
48         if(distancialineal(Origen, a) > distancialineal(Origen, b)){
49              return a;
50         }else{
51              return b;
52         }
53    }

55    procedure main()
56    {
57         writeln(masdistante(2, -3));
58         writeln(masdistante(-2, -3));
59    }
```

—

Una última nota. En capítulos anteriornes, cuando todavía no sabíamos usar *if-then-else*, se definió una función para calcular el valor absoluto de un número real. Dicha función consistía en calcular la raíz cuadrada del párametro elevado al cuadrado. Esa implementación tenía un

problema: el ordenador trata los números reales como aproximaciones, y realizando esas operaciones se pierde precisión en el número. La forma de calcular el valor absoluto que acabamos de ver es mucho mejor (aunque aquí se esté calculando para enteros, se podría hacer de forma similar para reales). Aquí, únicamente se le cambia el signo al número, por lo que no habría pérdida de precisión. Igualmente, una implementación basada en un *if-then-else* como

```
if(x < 0){
      return -x;
}else{
      return x;
}
```

tampoco provocaría una pérdida de precisión en el número real.

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

```
1 function espositivo(n: int): bool
2 {
3     if(n > 0){
4           return True;
5     }else{
6           return False;
7     }
8 }
```

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

```
1 function espositivo(n: int): bool
2 {
3     return n > 0;
4 }
```

Ten esto presente. Dado que los programas se hacen poco a poco es fácil acabar escribiendo código demasiado ineficaz. No pasa nada; pero cuando vemos código mejorable, debemos mejorarlo.

Otro caso típico es aquel en que se realiza la misma acción en todas las ramas de una estructura de ejecución condicional. Por ejemplo, mira el siguiente código. En este código aparece de nuevo *writeeol*, que simplemente sirve para escribir un final de línea. Dicho de otra forma, sirve para acabar con la línea que se está escribiendo y comenzar una nueva. El efecto de escribir un dato usando *write* seguido de *writeeol* es el mismo que el de escribir el dato usando *writeln*.

```
1  if(A > B){
2      write("el numero ");
3      write(A);
4      write(" es mayor que ");
5      write(B);
6      writeeol();
7  }else{
8      write("el numero ");
9      write(A);
10     write(" es menor o igual que ");
11     write(B);
12     writeeol();
13 }
```

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:

```
1  write("el numero ");
2  write(A);
3  if(A > B){
4      write(" es mayor que ");
5      write(B);
6      writeeol();
7  }else{
8      write(" es menor o igual que ");
9      write(B);
10     writeeol();
11 }
```

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  write("el numero ");
2  write(A);
3  if(A > B){
4      write(" es mayor que ");
5  }else{
6      write(" es menor o igual que ");
7  }
8  write(B);
9  writeeol();
```

¿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  write(cardigito16(valordigito(Numero, 3, 16)));
2  write(cardigito16(valordigito(Numero, 2, 16)));
3  write(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: int): char
```

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 restar 10 al valor entero, y hacer lo mismo que antes, pero desde el carácter `'A'`. Por ejemplo, para el valor 12, devolveríamos el carácter correspondiente al código de la letra `'A'` mas 2, esto es, la `'C'`:

```
1  /*
2   *  Caracter para un digito en base 16.
3   */
4  function cardigito16(valor: int): char
5  {
6      if(valor >= 0 and valor <= 9){
7             return char(int('0') + valor);
8      }else{
9             return char(int('A') + valor - 10);
10     }
11 }
```

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 (por ahora, siempre decimos que el día es válido):

|fechaok.p|

```
1     /*
2      *    Es valida una fecha?
3      */

5     program fechaok;

7     function esannovalido(anno: int): bool
8     {
9          return anno >= 1 and anno < 3000;
10    }
```

```
12    function esmesvalido(mes: int): bool
13    {
14         return mes >= 1 and mes <= 12;
15    }

17    /*
18     *   Ojo! Falta programar esta funcion!
19     */
20    function esdiavalido(dia: int): bool
21    {
22         return True;
23    }

25    function esfechavalida(dia: int, mes: int, anno: int): bool
26    {
27         return esannovalido(anno) and esmesvalido(mes) and esdiavalido(dia);
28    }


31    procedure main()
32    {
33         writeln(esfechavalida(1, 1, 1970));
34         writeln(esfechavalida(29, 2, 1970));
35    }
—
```

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: int, mes: int, anno: int): bool
```

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. Además, necesitamos una función auxiliar que nos diga si el año en cuestión es bisiesto o no. Pero esa ya la teníamos, o casi.

```
1  function esbisiesto(anno: int): bool
2  {
3      return anno % 4 == 0 and (not (anno %  100 == 0) or anno %  400 == 0);
4  }

6  function esdiavalido(dia: int, mes: int, anno: int): bool
7  {
8      switch(mes){
9      case 1, 3, 5, 7, 8, 10, 12:
10           return dia >= 1 and dia <= 31;

11     case 4, 6, 9, 11:
12           return dia >= 1 and dia <= 30;
```

```
13      case 2:
14              if(esbisiesto(anno)){
15                      return dia >= 1 and dia <= 29;
16              }else{
17                      return dia >= 1 and dia <= 28;
18              }

19      default:
20              return False;
21      }
22  }
```

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.p

```
1       /*
2        *    Es valida una fecha?
3        */

5       program fechaok;

7       function esannovalido(anno: int): bool
8       {
9            return anno >= 1 and anno < 3000;
10      }

12      function esmesvalido(mes: int): bool
13      {
14           return mes >= 1 and mes <= 12;
15      }

17      function esbisiesto(anno: int): bool
18      {
19           return anno % 4 == 0 and (not (anno % 100 == 0) or anno % 400 == 0);
20      }

22      function maxdiames(mes: int, anno: int): int
23      {
24           switch(mes){
25           case 1, 3, 5, 7, 8, 10, 12:
26                return 31;
27           case 4, 6, 9, 11:
28                return 30;
29           case 2:
30                if(esbisiesto(anno)){
31                     return 29;
32                }else{
33                     return 28;
34                }
35           default:
36                return 0;
37           }
38      }
```

```
40    function esdiavalido(dia:int, mes: int, anno: int): bool
41    {
42         return dia >= 1 and dia <= maxdiames(mes, anno);
43    }

45    function esfechavalida(dia: int, mes: int, anno: int): bool
46    {
47         return esannovalido(anno) and esmesvalido(mes) and
48              esdiavalido(dia, mes, anno);
49    }

51    procedure main()
52    {
53         writeln(esfechavalida(1, 1, 1970));
54         writeln(esfechavalida(29, 2, 1970));
55    }
—
```

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 Picky 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 1 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 *write* 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 *write*) produciría un efecto visible de forma externa (el mensaje). A eso se le denomina **efecto lateral** (Ver figura 5.1).

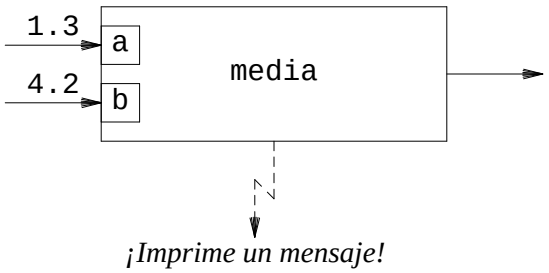


Figura 5.1: *En un efecto lateral una función provoca un cambio visible desde fuera.*

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 si la queremos ver como una función matemática! Un fragmento de código como

```
if(media(a, b) == 2){
      ...
}
```

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){
       ...
}
if(media(a, b) == 3){
       ...
}
```

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)`’’.

Además de las funciones, 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 conocemos varios procedimientos: *write*, *writeln* y *writeeol* son procedimientos.

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.* 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. Las variables se declaran entre la cabecera del subprograma y la llave que abre su cuerpo. Por ejemplo,

```
numero: int;
```

declara una variable llamada *numero* de tipo *int*. 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).

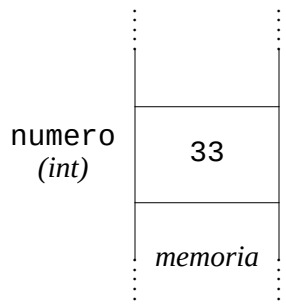


Figura 5.2: *Una variable es un nombre para una zona de memoria que guarda un valor de un tipo.*

Como nombres de variable vamos a utilizar identificadores que están escritos en minúsculas, para distinguirlos de constantes. 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*. Pero si usasemos 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:

```
write(numero);
```

Dicha sentencia **consulta** el valor de la variable *numero* y se lo pasa como argumento a *write*. Igualmente, si escribimos la expresión ‘‘`Pi*r**2`’’ entonces Picky 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 *int* (el valor escrito en la parte derecha de la asignación).

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. Por eso se dice que las variables son ‘‘L-values’’, o dicho de otro modo, que son elementos que pueden aparecer a la izquierda (la ‘‘L’’ es por ‘‘left’’) en una asignación.

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 main()
2      numero: int;
3  {
4      write("Voy a utilizar una variable");
5      writeeol();
6      numero = 3;
7      write("Le he dado un valor y vale ");
8      write(numero);
9      writeeol();
10 }
```

Al ejecutar el programa se procesa primero la zona de declaraciones entre la cabecera del subprograma y su cuerpo (las sentencias del subprograma que estan entre las llaves). 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, Picky ejecuta las sentencias del cuerpo del programa (líneas 3 a 10). 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 *int*. 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 *numero* que se utiliza en la sentencia *write(numero)*, que escribe el valor de la variable en la salida estándar.

Veamos otro ejemplo. Supongamos que tenemos el programa:

```
1  procedure main()
2      x: int;
3      y: int;
4  {
5      x = 3;
6      y = x + 1;
7      x = y / 2;
8  }
```

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 *y* no lo está. 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. El símbolo `=` que vemos en la sentencia, es un símbolo de asignación que significa que a la parte izquierda se le da el valor de la parte derecha. No significa que la parte izquierda sea igual que la parte derecha. No hay que confundirlo con

```
x == x + 1
```

que es una expresión de tipo *bool* cuyo valor es siempre *False* (puesto que *x* sólo puede tener un único valor a la vez).

Pero veamos lo que hace

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

Primero Picky 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 *int* y de un literal ‘‘`1`’’ de tipo *int*. 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. ¡Eso es justo lo que hace ‘‘`=`’’! 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**. Es una sentencia popular en variables destinadas a contar cosas (llamadas **contadores**).

Antes de seguir hay una cosa muy importante que es preciso decir. 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. Dicho de otro modo, las variables no deben mentir: si una variable es *numcartas*, debería tener siempre como valor el número de cartas que tenemos, ni una mas, ni una menos; sea cual sea la línea de código que estemos considerando.

5.4. Más sobre variables

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

```
c: char;

 ...

c = 'A';
```

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

```
c = char(int(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

```
int(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 hacemos una conversión explícita a *char* para obtener el carácter con dicha posición, obtenemos el valor `'B'` de tipo *char*. Este valor se lo hemos asignado a *c*, con lo que *c* pasa a tener el valo `'B'`. Una forma más directa habría sido utilizar

```
c = succ(c);
```

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

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 *read*. Por ejemplo, el siguiente programa lee un número y luego escribe el número al cuadrado en la salida.

cuadrado.p

```
1     /*
2      *    Elevar un numero al cuadrado.
3      */

5     program cuadrado;

7     procedure main()
8           numero: int;
9     {
10          read(numero);
11          writeln(numero ** 2);
12    }
```

La sentencia *read* lee desde teclado un valor para la variable que se pasa como argumento. Osea, un entero en este caso. Una vez hecho eso, *read* asigna ese valor a la variable en cuestión. A partir de la línea 10 del programa tendremos en la variable *numero* el número que haya escrito el usuario.

Hay otra sentencia para leer: *readln*. Esta es igual que *read* salvo porque, tras efectuar la lectura, se salta todo lo que exista hasta el final de la línea en la entrada.

Recuerda que algunos procedimientos y funciones predefinidos (*built-in*) en Picky aceptan argumentos de distintos tipos (por ejemplo *write*, *writeln*, y otros). Estos subprogramas son **polimórficos**. Ese es el caso de *read* y *readln*. Por ejemplo, si *x* es de tipo *char*

```
read(x);
```

lee un carácter, pero si *x* es de tipo *int*, lee un entero. En nuestros programas no podemos definir subprogramas que actúen de esta forma. Eso sólo lo pueden hacer algunos *built-in* de Picky.

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 *read* 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.p

```
1     /*
2      *    Escribir dos numeros ordenados.
3      */

5     program ordenar2;
```

```
7     procedure main()
8           a: int;
9           b: int;
10    {
11          read(a);
12          read(b);

14          if(a < b){
15                writeln(a);
16                writeln(b);
17          }else{
18                writeln(b);
19                writeln(a);
20          }
21    }
—
```

5.6. Procedimientos

Una función no puede modificar los argumentos que se le pasan, pero es muy útil hacer subprogramas (como *read*) que pueden modificar los argumentos o que pueden tener otros efectos laterales. Esos subprogramas se llaman **procedimientos** y en realidad ya los conocemos. En Picky, el programa principal es un procedimiento. Tanto *read* como *write* son procedimientos. También hemos visto programas en los que hemos definido un nuevo procedimiento, por ejemplo, en el programa *cuadrado* mostrado un poco más arriba.

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, *write* es en realidad un nombre para la acción o acciones que tenemos que realizar para escribir un valor. Igualmente, *read* 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.

cuadrado.p

```
1     /*
2           Elevar un numero al cuadrado.
3      */

5     program cuadrado;

7     procedure main()
8           numero: int;
9     {
10          read(numero);
11          writeln(numero ** 2);
12          read(numero);
13          writeln(numero ** 2);
14    }
—
```

Como podemos ver, el cuerpo del programa realiza dos veces la misma secuencia:

```
    read(numero);
    writeln(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.p

```
1     /*
2      *    Elevar dos numeros al cuadrado.
3      */

5     program cuadrados;

7     procedure cuadrado()
8           numero: int;
9     {
10          read(numero);
11          writeln(numero ** 2);
12    }

14    procedure main()
15    {
16          cuadrado();
17          cuadrado();
18    }
—
```

Aquí podemos ver que hemos definido un procedimiento llamado *cuadrado*. Esto es, nos hemos inventado un nombre para la acción consistente en imprimir el cuadrado de un número que se lee por teclado. Dicho nombre es *cuadrado*. Ahora, en el programa principal, podemos utilizar dicha acción sin mas que invocar su nombre (como el que invoca a un espíritu). ¡Podemos invocarlo cuantas veces queramos!

Las líneas 16 y 17 de nuestro programa invocan o llaman al procedimiento *cuadrado*. Por tanto decimos, por ejemplo, que la línea 16 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: el programa comienza ejecutando en su programa principal, en la línea 15. Cuando el flujo de control del programa alcanza la línea 16, Picky 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*. Cuando se ejecuten las sentencias del procedimiento *cuadrado*, el procedimiento termina (decimos que **retorna**) y se continúa con lo que se estaba haciendo (en este caso, ejecutar el programa principal). Después, en la línea 17, sucede lo mismo. En ese punto se llama al procedimiento *cuadrado* y se ejecutan las sentencias de dicho procedimiento. Cuando se acaban las sentencias del procedimiento, se retorna. Como esta es la última sentencia del procedimiento principal, el programa acaba su ejecución. 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 *main* 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

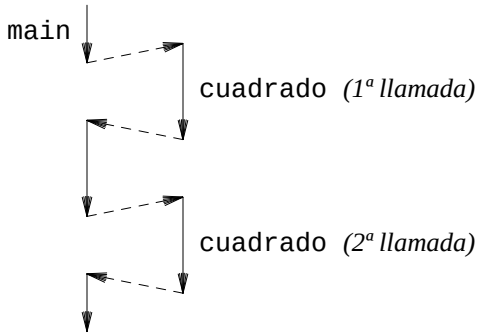


Figura 5.3: *Flujo de control del programa* `cuadrados` *con dos llamadas a procedimiento*

ejecute el programa (no existe hasta que lo ejecutamos).

- 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 *read* y cuando *cuadrado* llama a *write*. La figura 5.4 muestra esto. Las llamadas a procedimiento se pueden **anidar** sin problemas (un procedimiento puede llamar a otro y así sucesivamente).

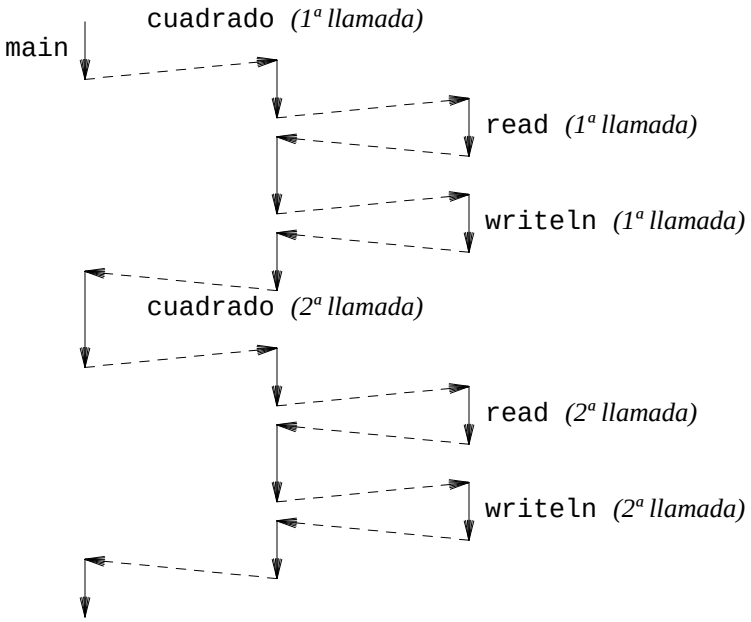


Figura 5.4: *Llamadas a procedimiento anidadas*

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.

En Picky existe un procedimiento predefinido llamado *stack* que sirve para imprimir el estado de la pila. Esto es muy útil para depurar un programa (para cazar bugs, o eliminar errores que hemos cometido). Por ejemplo, si modificamos el procedimiento cuadrado así:

```
7     procedure cuadrado()
8           numero: int;
9     {
10          read(numero);
11          writeln(numero ** 2);
12          stack();
13    }
```

Cuando se ejecute la última sentencia (línea 12) del procedimiento *cuadrado*, se escribirá en la salida el estado de la pila en ese momento. Por ejemplo:

```
stack trace at:
cuadrado() pid 0    pc 0x00001b cuadrados.p:12
       local variables:
       numero = 4

called from:
main() pid 1  pc 0x00001f cuadrados.p:17
```

La información de la pila nos resultará muy útil en el futuro, cuando intentemos arreglar nuestros programas. Básicamente, aquí se nos está diciendo que (en el momento de la llamada a *stack*) el programa está ejecutando *cuadradro* (justo en la línea 12 del fichero fuente `cuadrados.p`). Se nos dice además que en esa llamada *cuadrado* tiene una variable llamada *numero* con un valor de 4; y que a *cuadrado* se lo llamó desde el procedimiento *main* (justo en la línea 17 de `cuadrados.p`). Por ahora, no necesitamos saber más sobre esto.

5.7. Parámetros

Hemos visto cómo se producen las llamadas a procedimiento. Pero hemos omitido de la discusión el paso de parámetros.

Podemos declarar parámetros en los procedimientos 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.p

```
1     /*
2      *    Sumados numeros.
3      */

5     program suma;

7     procedure escribeentero(n: int)
8     {
9           write("El valor es: ");
10          writeln(n);
11    }
```

```
13    procedure main()
14          numero1: int;
15          numero2: int;
16          resultado: int;
17    {
18          writeln("introduce dos numeros: ");
19          read(numero1);
20          read(numero2);
21          resultado = numero1 + numero2;
22          escribeentero(resultado);
23    }
—
```

Las líneas 7 a 11 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 22 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 22).

Cuando el procedimiento llega a su fin, 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, nada varía con respecto a lo ya dicho: el parámetro del procedimiento es una variable distinta a la variable empleada en el programa principal, una copia ¡Incluso si tienen el mismo nombre! Piensa que todo esto ya lo conoces: es exactamente lo mismo que sucede con los parámetros de las funciones.

Cuando declaramos parámetros de este modo, se copia el valor del argumento al parámetro, justo al principio de la llamada a subprograma. Picky permite la modificación del valor del parámetro, pero 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. Esto es, podemos utilizar este mecanismo para declarar e implementar un procedimiento como *write*, pero no podemos utilizarlo para implementar un procedimiento como *read*.

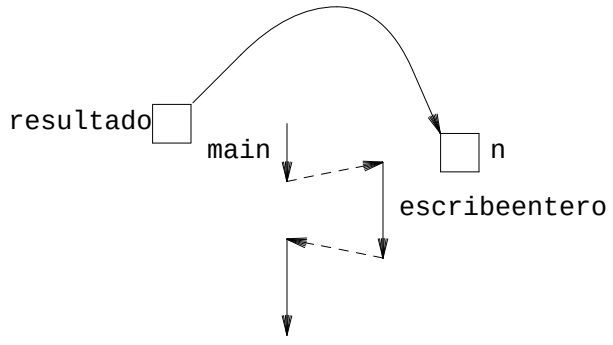


Figura 5.5: *Paso de parámetros por valor: se copia el valor del argumento al llamar.*

Pero veamos ahora un procedimiento que incrementa el valor de una variable. El siguiente programa imprime 2 en su salida estándar.

incr.p

```
1     /*
2      *    Incrementa un numero.
3      */

5     program incr;

7     procedure incrementar(ref n: int)
8     {
9           n = n + 1;
10    }

12    procedure main()
13          numero: int;
14    {
15          numero = 1;
16          incrementar(numero);
17          writeln(numero);
18    }
```

Comparando el procedimiento *incrementar* con el procedimiento *escribeentero* programado antes, podrá verse que la principal diferencia es que el parámetro especificado en la cabecera tiene la palabra reservada *ref* antes del nombre del parámetro.

Cuando un parámetro se declara como *ref*, dicho parámetro corresponde en realidad a la variable suministrada como argumento. Esto quiere decir que *el procedimiento es capaz de modificar el argumento.* En nuestro programa de ejemplo, es como si el parámetro *n* fuese lo mismo que *numero* durante la llamada de la línea 16. La figura 5.6 muestra lo que sucede ahora durante la llamada: *n* se comporta como otro nombre para la variable *numero* pasada como argumento.

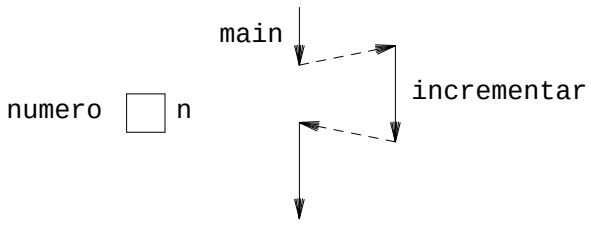


Figura 5.6: *Paso por referencia: durante la llamada el parámetro es lo mismo que el argumento*

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.

Dado que *n* se refiere en realidad al argumento que ha recibido (esto es, a *numero*) 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**. 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).

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. Si el paso es por valor, son variables que están declaradas automáticamente al comenzar la ejecución del procedimiento del que son parámetro, y son una copia del argumento que se ha pasado en la llamada. Por tanto dejan de existir cuando el procedimiento (o función) termina.

Si el paso es por referencia, un parámetro es en realidad la variable pasada como argumento, no una copia. Cuando se termina un procedimiento, todo lo que se ha hecho sobre el parámetro tiene efecto en la variable pasada como argumento en la llamada.

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 y qué variables se pasan por referencia.

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 (por que corresponda una foto digital, por ejemplo) se suele utilizar paso de parámetros por referencia incluso si sólo se quiere consultar el parámetro. La razón es que, en general, el paso de parámetros por referencia evita hacer copia de los argumentos, y por tanto es más rápido y consume menos memoria.

En este curso seguiremos esta costumbre, de tal forma que en general si un parámetro se considera que tiene un tamaño excesivo se empleará *ref* 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 tener parámetros por referencia.*

5.8. Variables globales

A las variables declaradas fuera de un subprograma se las denomina **variables globales** o **externas**. En este curso **no se permite el uso variables externas a un subprograma** (procedimiento o función).

El compilador de Picky, por omisión, no permite la declaración de variables globales. Un programa con variables globales no compila, a no ser que se ejecute el compilador con la opción `-g`. Por ejemplo

```
; pick -g inct.p
```

Naturalmente, en nuestros programas 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.

Veamos como queda el programa anterior si usa una variable global (evidentemente, este programa está mal escrito):

```
1     /*
2      *    Incrementa  un numero.
3      *    Mal,  porque tiene una variable global.
4      *    Pero que muy mal! Para matarse, vamos!
5      */

7     program incr;

9     vars:
10          numero: int;    /* var. global */
```

```
12    procedure incrementar()
13    {
14          numero = numero + 1;
15    }

17    procedure main()
18    {
19          numero = 1;
20          incrementar();
21          writeln(numero);
22    }
—
```

En este caso, el procedimiento *incrementar* utiliza directamente la variable global *numero*. Entonces, ese procedimiento sólo sirve 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.

Las variables globales son útiles en ciertos casos (siempre que las use un programador experimentado) que quedan fuera del ámbito de este curso de programación. Y por cierto, los programadores experimentados intentan evitar el uso de variables globales.

5.9. Visibilidad y ámbito

Consideremos el siguiente programa.

ambitos.p

```
1     /*
2      *   Programa absurdo que ademas viola las normas
3      *   para ayudar con la comprensión del uso de variables.
4      *   Esta mal, pero que muy mal.
5      */

7     program ambitos;

9     procedure proc1(n: int, ref n2: int)
10          n: int;
11    {
12          n2 = n ** 2;
13    }
```

```
15    procedure proc2(n: int)
16    {
17         proc1(3, n);
18         writeln(n);
19    }

21    procedure main()
22         n: int;    /* (1) */
23    {
24         read(n);
25         proc2(n);
26    }
—
```

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 final del programa principal, luego existe durante todo el programa. La llamada a *read* 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 dejará de existir cuando *proc2* llegue a su final.

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. El parámetro *n2* es *por referencia* , por tanto, se refiere al argumento que se pasó (que es la variable *n* de *proc2*). Por tanto, lo que se haga sobre el parámetro *n2* de *proc1* tiene efecto sobre la variable *n* de *proc2*. Después de la cabecera de *proc1*, se declara una variable llamada *n*. Estas tres variables dejan de existir cuando *proc1* termina de ejecutar y el flujo de control retorna a *proc2*.

En *proc1* tenemos un problema. La variable local *n* de *proc1* **oculta** el parámetro del mismo nombre, dado que no puede haber dos variables que se llamen igual en el mismo bloque de sentencias. De hecho, esta declaración no se puede hacer en Picky: este programa no compila. Pero en otros lenguajes de programación sí compilaría, y el identificador *n* se referiría dentro de ese procedimiento a la variable local y no al parámetro. El compilador de Picky nos protege ante este tipo de errores. Pero si el lenguaje nos permitiese hacer eso, estaríamos ocultando un parámetro y perdiéndolo, con lo que *proc1* sería erróneo.

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) a la parte del programa en que dicha variable existe. Las variables declaradas en un procedimiento tienen como ámbito la parte del programa que va 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** (y no se pueden usar durante este curso).

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 principal. Esta variable sólo es visible en el cuerpo del programa principal.

Recordamos de nuevo que declarar variables que oculten parámetros o variables no es una buena práctica y se debe evitar, aunque el lenguaje que estemos usando lo permita.

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:

ordenarpuntos.p

```
1   /*
2    * Ordenar puntos segun sus distancias al origen y escribir las distancias.
3    */

5   program ordenarpuntos;

7   ...

9   procedure main()
10      a: int;
11      b: int;
12  {
13      leerpunto(a);
14      leerpunto(b);
15      ordenar(a, b);
16      escribirdistancia(a);
17      escribirdistancia(b);
18  }
```

Hemos declarado dos variables locales en el procedimiento principal, *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 *int* 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, cuyo valor no se sabrá hasta que el programa ejecute y se lea de la entrada.

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 *escribirdistancia(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 pasarse por referencia. Por lo tanto, podemos programar:

```
1  procedure leerpunto(ref punto: int)
2  {
3      write("Introduce la posicion del punto: ");
4      read(punto);
5  }
```

El procedimiento *read* estará declarado de un modo similar, por eso es capaz de modificar *punto* en la llamada de la línea 4.

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  procedure ordenar(ref a: int, ref b: int)
2  {
3      if(distancialineal(Cero, a) > distancialineal(Cero,b)){
4             intercambiar(a, b);
5      }
6  }
```

Ambos parámetros están declarados como *ref* 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 *ref*, digamos:

```
procedure intercambiar(ref a: int, ref b: int)
```

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(ref a: int, ref b: int)
2      aux: int;
3  {
4      aux = a;
5      a = b;
6      b = aux;
7  }
```

Como vemos, la interfaz (la cabecera del procedimiento) sigue siendo la misma: recibe dos puntos y los ordena, dejándolos ordenados a la salida. Pero declaramos una variable local *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 más. El programa completo nos ha resultado como sigue.

ordenarpuntos.p

```
1    /*
2     *   Ordenar puntos segun sus distancias al origen y escribir las distancias.
3     */

5    program ordenarpuntos;

7    consts:
8         Cero = 0;
```

```
function distancialineal(a: int, b: int): int
{
     if(a > b){
          return a - b;
     }else{
          return b - a;
     }
}

procedure leerpunto(ref punto: int)
{
     write("Introduce la posicion del punto: ");
     read(punto);
}

procedure escribirdistancia(punto: int)
{
     write("punto : ");
     write(punto);
     write(" distancia: ");
     writeln(distancialineal(Cero, punto));
}

procedure intercambiar(ref a: int, ref b: int)
     aux: int;
{
     aux = a;
     a = b;
     b = aux;
}

procedure ordenar(ref a: int, ref b: int)
{
     if(distancialineal(Cero, a) > distancialineal(Cero,b)){
          intercambiar(a, b);
     }
}

procedure main()
     a: int;
     b: int;
{
     leerpunto(a);
     leerpunto(b);
     ordenar(a, b);
     escribirdistancia(a);
     escribirdistancia(b);
}
```

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.

ec2grado.p

```
1     /*
2      *    Resolver una ecuacion de segundogrado.
3      */
4     program ec2grado;

6      ...

8     procedure main()
9          a: float;
10         b: float;
11         c: float;
12         sol1: float;
13         sol2: float;
14    {
15         leerecuacion(a, b, c);
16         if(existesolucionreal(a, b, c)){
17              calcularsolucion(a, b, c, sol1, sol2);
18              imprimirsolucion(sol1, sol2);
19         }else{
20              writeln("No hay solucion real");
21         }
22    }
```

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. 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.

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).

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.p

```
1     /*
2      *    Resolver una ecuacion de segundo grado.
3      */
4     program ec2grado;

6     procedure leerecuacion(ref a: float, ref b: float, ref c: float)
7     {
8          write("Escribe los coeficientes a, b y c: ");
9          read(a);
10         read(b);
11         read(c);
12    }

14    function existesolucionreal(a: float, b: float, c: float): bool
15    {
16         return a != 0.0 and b ** 2.0 >= 4.0 * a * c;
17    }

19    procedure calcularsolucion(a: float,
20                               b: float,
21                               c: float,
22                               ref sol1: float,
23                               ref sol2: float)
24         discriminante: float;
25    {
26         writeln(b);
27         writeln(-b);
28         discriminante = b ** 2.0 - 4.0 * a * c;
29         writeln((-b) + sqrt(discriminante));
30         sol1 = ((-b) + sqrt(discriminante)) / (2.0 * a);
31         sol2 = ((-b) - sqrt(discriminante)) / (2.0 * a);
32    }

34    procedure imprimirsolucion(sol1: float, sol2: float)
35    {
36         write("Solucion:");
37         if(sol1 == sol2){
38              write("x = ");
39              write(sol1);
40         }else{
41              write("x1 = ");
42              write(sol1);
43              write(" x2 = ");
44              write(sol2);
45         }
46         writeeol();
47    }
```

```
49    procedure main()
50          a: float;
51          b: float;
52          c: float;
53          sol1: float;
54          sol2: float;
55    {
56          leerecuacion(a, b, c);
57          if(existesolucionreal(a, b, c)){
58                calcularsolucion(a, b, c, sol1, sol2);
59                imprimirsolucion(sol1, sol2);
60          }else{
61                writeln("No hay solucion real");
62          }
63    }
```

—

Este programa en realidad tiene un error, la comparación

```
sol1 == sol2
```

está comparando dos números reales. Como ya se ha explicado en capítulos anteriores, 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 de hacerse 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 hecho en el ejercicio 6 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 *ADADADADADA* 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 Picky. Esto es, tipos que forman parte del mundo según lo ve Picky: *int*, *float*, *char* y *bool*.

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 Picky 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){
      ...
}
```

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.

Picky permite **definir nuevos tipos** para inventarnos mundos que no existen inicialmente en Picky (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.

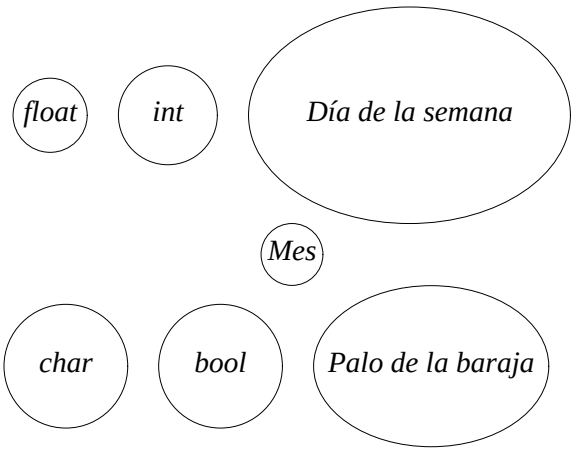


Figura 6.1: *Además de enteros, caracteres, reales y booleanos podemos inventar nuevos mundos.*

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**. En un tipo enumerado los elementos (los literales) son enumerables y se le puede asignar un número natural a cada elemento.

En Picky, una declaracion de un nuevo tipo de datos define dicho tipo y le da un nombre. Las declaraciones de tipos deben estar en una sección *types*. Por ejemplo:

```
types:
      TipoDiaSem = (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
      TipoMes = (Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov, Dic);
```

La primera línea indica que siguen varias definiciones de tipo. La segunda línea define un nuevo tipo enumerado llamado *TipoDiaSem*, para representar los días de la semana. La tercera línea define un nuevo tipo enumerado para los meses del año. Esta otra podría utilizarse para el valor de una carta:

```
      TipoValor = (Uno, Dos, Tres, Cuatro, Cinco,
                   Seis, Siete, Sota, Caballo, Rey);
```

Naturalmente podríamos también definir:

```
      TipoPaloBaraja = (Bastos, Oros, Copas, Espadas);
```

O tal vez queramos manipular dígitos de números romanos:

```
      TipoDigitoRomano = (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 (valores constantes) 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 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 *int*. De hecho, *Vie* estará representado dentro del ordenador por un número entero, por ejemplo el 4 o el 5. Pero eso no quiere decir que sea un *int*.

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:

```
consts:
      UnBuenDia = 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 vemos en los ejemplos. Los identificadores para los elementos del tipo también estarán capitalizados del mismo modo, con cada 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 anteriormente para definir un dígito romano son pésimos. Posiblemente fuese mejor utilizar algo como

```
TipoDigitoRomano = (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 (los que conocíamos ya 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  /*
2   *  Leer un caracter y escribir el siguiente.
3   */
4  program siguientechar;
5
6  procedure main()
7      c: char;
8  {
9      read(c);
10     c = succ(c);
11     writeln(c);
12 }
```

Pues bien, el siguiente programa lee un día de la semana, calcula el siguiente día y lo imprime. Todo esto también suponiendo que el día que ha leído no es el último:

```
1  /*
2   *  Leer un dia y escribir el siguiente.
3   */
4  program siguientedia;
5
6  types:
7      TipoDiaSem = (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
8
9  procedure main()
10     dia: TipoDiaSem;
11 {
12     read(dia);
13     dia = succ(dia);
14     writeln(dia);
15 }
```

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  /*
2   *  Leer un dia y escribir el siguiente.
3   */
4  program siguientedia;
5
6  types:
7      TipoDiaSem = (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
8
9  procedure main()
10     dia: TipoDiaSem;
11 {
12     read(dia);
13     if(dia == Dom){
14            dia = Lun;
15     }else{
16            dia =  succ(dia);
17     }
18     writeln(dia);
19 }
```

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, los operadores de comparación también está definidos. El orden de los literales del tipo enumerado es precisamente el orden utilizado en su declaración (de menor a mayor).

Las funciones predefinidas de Picky llamadas *succ* y *pred* aceptan cualquier tipo enumerado. En general, todo lo que podemos hacer con *char* 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).

Podemos conseguir el entero que indica la posición de un valor dentro un tipo enumerado de la misma forma que lo hacemos con los caracteres. En el siguiente ejemplo se asigna a una variable de tipo *int* la posición del valor *Dom* en el tipo *TipoDiaSemana*:

```
posdom = int(Dom);
```

Igualmente, podemos obtener el valor correspondiente a una posición dada en el tipo enumerado:

```
dia = TipoDiaSem(1); /* la posicion 1 es el martes, Mar */
```

Otro detalle importante es que podemos leer días de la entrada estándar y escribirlos en la salida estándar. Únicamente es cuestión de usar

```
read(dia);
```

y

```
write(dia);
```

de la misma forma que lo hemos hecho hasta ahora. Si lo que se lee de la entrada no corresponde con un valor del tipo en cuestión, tendremos un error de ejecución.

6.2. Mundos paralelos y tipos universales

En Picky podemos definir tipos a partir de otros tipos. Cuando hacemos esto, una variable del nuevo tipo definido puede adquirir cualquiera de los valores disponibles para el tipo original, pero **no** es compatible con él. Por ejemplo:

```
types:
       TipoElemento = char;
       TipoManzanas = int;
       TipoPeras = int;
```

Aquí se han declarado tres nuevos tipos de datos: *TipoElemento*, *TipoManzanas* y *TipoPeras*. El primer tipo tiene los mismos valores que el tipo de datos *char*. Pero no se puede mezclar con los *char*.

Los tipos *TipoManzanas* y *TipoPeras* son similares a *int*. Una variable *p* de tipo *TipoPeras* puede tomar como valor un entero cualquiera, lo mismo que *int*. Pero no es de tipo *int*, ni es compatible con él. Pasaría lo mismo con una variable *m* de tipo *TipoManzanas*. Aunque puedan tomar los mismos valores numéricos, no podremos mezclar peras con manzanas, por ejemplo:

```
p: TipoPeras;
m: TipoManzanas;
 ...
writeln(p + m);      ¡incorrecto!
```

¡Esas variables son de tipos distintos y no se pueden mezclar!

De hecho, tampoco pueden operar con variables de tipo *int*. Suponiendo que la variable *i* es de tipo *int*, la siguiente sentencia:

```
m = i;               ¡incorrecto!
```

no compila, ya que *m* es de tipo *TipoManzanas* e *i* es de tipo *int*, y en Picky no se pueden mezclar tipos.

No obstante, en Picky las constantes y los literales son de **tipos universales**. Esto significa que las constantes y los literales se pueden mezclar con los tipos de su misma ‘‘categoría’’. Por ejemplo, el literal 2 se puede mezclar con cualquier tipo de la categoría de los números enteros, ya sea *int*, *TipoManzana* o cualquier otro tipo basado en los números enteros. Eso es porque en realidad el literal *2* es de tipo **entero universal**. Dicho de otra forma, es compatible con cualquier tipo de datos que se represente con números enteros (entre ellos, el tipo *int*). Pero eso sí, no es compatible con un tipo de datos que se represente con números reales, caracteres, o días de la semana. Por ejemplo, podríamos tener esta sentencia:

```
m = 2 * m;           ¡correcto!
```

Dicha sentencia nos doblara el número de manzanas que representa *m*, que es de tipo *TipoManzanas*. Podemos ver el literal *2* como el número dos que manejamos los humanos en el día a día, en nuestro lenguaje natural. Si se opera con *TipoManzanas*, el resultado de la expresión es de tipo *TipoManzanas*. Es como si te dijesen en el mercado: *hoy hay oferta de 2x1, multiplica las manzanas de tu bolsa por dos*. El resultado serían manzanas, no números. Ese es el sentido de un tipo universal. Ten en cuenta que esto no se podría hacer (siendo *i* de tipo *int*):

```
i = 2;
m = i * m;           ¡incorrecto!
```

Es ilegal, porque *int* (que no es el tipo universal entero) es un tipo distinto a *TipoManzanas*.

6.3. Subrangos

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 Picky tenemos **subrangos** para expresar este concepto.

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 Picky como sigue:

```
types:
       TipoDiaSem = (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
       TipoDiaLab = TipoDiaSem Lun..Vie;
```

A partir de esta declaración 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 subrango: la lista de días comprendidos entre *Lun* y *Vie* incluyendo ambos extremos del intervalo. Por cierto, al tipo empleado para definir el subrango se le llama **tipo base** del subrango. 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 variables de ambos tipos entre sí (con cuidado de que no se salgan del rango, claro). Por ejemplo, si la variable *hoy* es de tipo *TipoDiaSem*, podríamos hacer esto:

```
hoy = Mie;
diareunion = hoy;    ¡correcto!
```

Eso sí, si en algún momento intentamos ejecutar algo como

```
diareunion = Dom;    ¡incorrecto! se sale de rango
```

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 subconjuntos de los enteros. Por ejemplo, para los naturales y para los días del mes podríamos definir

```
types:
       TipoNatural = int 0..Maxint;
       TipoDiaDelMes = int 1..31;
```

El tipo *TipoNatural* es el rango de los números naturales, del 0 al número entero máximo representable en Picky. El otro tipo del ejemplo se puede usar para los días del mes. De ese modo, si utilizamos variables de tipo *TipoDiaDelMes* en lugar de variables de tipo *int*, el lenguaje nos protegerá comprobando que cada vez que asignemos un valor a alguna de estas variables el valor está dentro del rango permitido. Imagina que nuestro programa está mal, y decide planificar una cita el día 42 de marzo. Si usamos este tipo subrango, el programa fallará en ejecución (ya que ese número no está en el subrango). Si usamos una variable de tipo *int*, el programa seguiría ejecutando en un estado incorrecto, y realizaría operaciones sin sentido. Es mejor que falle cuanto antes, para poder detectar el problema con más facilidad.

La posibilidad de definir subrangos está presente para todos los tipos enumerados. Luego podemos definir subtipos como siguen:

```
TipoLetraMay = char  'A'..'Z';
TipoLetraMin = char 'a'..'z';
TipoDigito = char '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

```
switch(c){
case 'A'..'Z':
      ...
case '0'..'9':
      ...
}
```

6.4. 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,
             ref sol1: float,
             ref sol2: float)
```

cuando en realidad habría sido mucho más apropiado poder declarar:

```
procedure calcularsolucion(ec: TipoEcuacion, ref sol: 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 *TipoValor* x *TipoPalo*.

En Picky 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:

```
types:
      TipoValor = int 1..10;
```

Igualmente podríamos definir un tipo enumerado para el palo de la baraja al que pertenece una

carta:

```
TipoPalo = (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.

```
TipoCarta = record
{
        valor: TipoValor;
        palo: TipoPalo;
};
```

Como puede verse, la declaración utiliza la palabra reservada *record* (de ahí el nombre de estos tipos de datos) e incluye entre las llaves una declaración para cada elemento de la tupla. En este caso, una carta está formada por algo llamado *valor* (de tipo *TipoValor*) 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 *TipoValor* 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:

```
types:
        /*
         *      a * x**2 + b * x + c == 0
         */
        TipoEcuacion = record
        {
                a: float;
                b: float;
                c: float;
        };
```

En este caso una ecuación es una terna ordenada 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 *TipoValor* y otra después de tipo *TipoPalo*. La figura 6.2 muestra el aspecto de dos *records* en la memoria del ordenador.

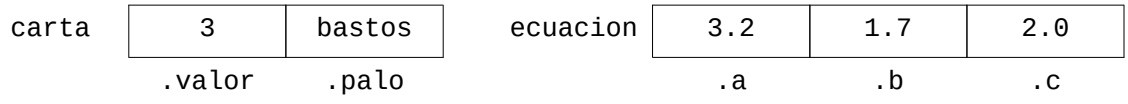


Figura 6.2: *Aspecto de los records de los ejemplos en la memoria del ordenador.*

¿Y cómo podemos utilizar una variable de tipo *TipoCarta*? En general, igual que cualquier otra variable. Por ejemplo:

```
carta1: TipoCarta;
carta2: TipoCarta;
        ...
carta1 = carta2;    /* asigna el valor de carta2 a carta1 */
```

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 *TipoValor*. 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:

```
consts:
      AsCopas = TipoCarta(1, Copas);
```

Como puede verse, hemos inicializado la constante escribiendo el nombre del tipo y, entre paréntesis, una expresión para cada campo del registro, en el orden en que están declarados los campos. El ‘‘1’’ corresponde a *AsCopas.valor* y ‘‘*Copas*’’ corresponde a *AsCopas.palo* (como valores iniciales, durante la definición de la constante). La tupla escrita a la derecha de la definición 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(ref carta: TipoCarta)
2 {
3       read(carta.valor);
4       read(carta.palo);
5 }
```

Dado que ambos campos son de tipos elementales (no son de tipos compuestos de varios

elementos, como los records) podemos leerlos directamente.

6.5. 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* (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.*

6.6. Geometría

Queremos manipular figuras geométricas en dos dimensiones. Para manipular puntos podríamos declarar:

```
types:
       TipoPunto = record
       {
              x: int;
              y: int;
       };
```

Un punto es un par de coordenadas, por ejemplo, en la pantalla del ordenador. Podríamos declarar el origen de coordenadas como

```
consts:
       Origen = TipoPunto(0, 0);
```

lo que haría que *Origen.x* fuese cero y que *Origen.y* fuese cero.

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:

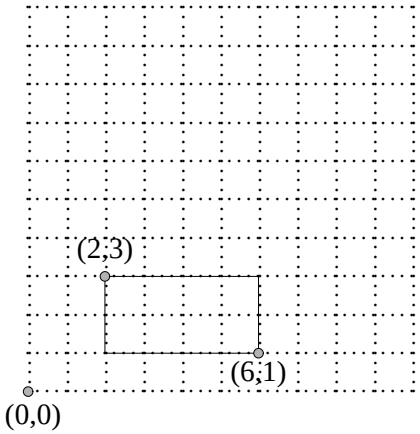


Figura 6.3: *Coordenadas en dos dimensiones incluyendo origen y un rectángulo.*

```
TipoRectangulo = record
{
       min: TipoPunto;
       max: TipoPunto;
};
```

Ahora, dado un rectángulo

```
r: TipoRectangulo;
```

podríamos escribir *r.min* para referirnos a su punto de arriba a la izquierda. O quizá *r.min.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:

```
TipoCirculo = record
{
       centro: TipoPunto;
       radio: int;
};
```

Las elipses son similares, pero tienen un radio menor y un radio mayor:

```
TipoElipse = record
{
       centro: TipoPunto;
       radiomin: int;
       radiomax: int;
}
```

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){
       ...
}
```

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 en 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;
```

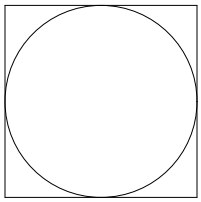


Figura 6.4: *Círculo inscrito en un cuadrado.*

Hemos supuesto que nuestro *cuadrado* es en realidad un rectángulo. ¿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 *bool*:

```
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 vamos a hacer programas con todo lo que hemos visto.

6.7. 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):

```
types:
       TipoComplejo = record
       {
              re: float;/* parte real */
              im: float;/* parte imaginaria */
       };
```

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 el procedimiento *leercomplejo* emplea paso de parámetros por referencia.

complejos.p

```
1     /*
2      *    Aritmetica de numeros complejos.
3      */

5     program complejos;

7     types:
8          TipoComplejo = record
9          {
10              re: float; /* parte real */
11              im: float; /* parte imaginaria */
12         };

14    function nuevocomplejo(re: float, im: float): TipoComplejo
15         c: TipoComplejo;
16    {
17         c.re = re;
18         c.im = im;
19         return c;
20    }

22    procedure leercomplejo(ref c: TipoComplejo)
23         re: float;
24         im: float;
25    {
26         write("Introduce parte real e imaginaria: ");
27         read(re);
28         read(im);
29         c = nuevocomplejo(re, im);
30    }

32    procedure escribircomplejo(c: TipoComplejo)
33    {
34         write(c.re);
35         write(" + ");
36         write(c.im);
37         write('i');
38    }

40    procedure main()
41         c: TipoComplejo;
42    {
43         leercomplejo(c);
44         escribircomplejo(c);
45         writeeol();
46    }
```

—

Antes de seguir, como de costumbre, debemos ejecutar el programa para ver si todo funciona. Y parece que sí.

```
; pick complejos.p
; out.pam
Introduce parte real e imaginaria: 3.4 2
3.400000 + 2.000000 i
```

Hemos de recordar que en Picky podríamos escribir un agregado como

```
c = TipoComplejo(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): TipoComplejo
{
       return nuevocomplejo(c1.re + c2.re, c1.im + c2.im);
}

function restar(c1: TipoComplejo, c2: TipoComplejo): TipoComplejo
{
       return nuevocomplejo(c1.re - c2.re, c1.im - c2.im);
}
```

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, pero rompe la abstracción puesto que vemos cómo está hecho un número complejo.

6.8. 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 main()
       carta: TipoCarta;
       valor: float;
{
       leercarta(carta);
       valor = valorcarta(carta);
       write("La carta vale ");
       writeln(valor);
}
```

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 *write* para escribir el valor.

Hemos hecho casi lo mismo con la carta. ¡Aunque Picky no tiene ni idea de lo que es una carta! Nos hemos inventado el tipo *TipoCarta*, que después tendremos que definir en la sección correspondiente, 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, *valorcarta* es una función que también nos hemos sacado de la manga para que nos devuelva el valor de 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:

```
TipoCarta = record
{
       valor: TipoValor;
       palo: TipoPalo;
};
```

Y ahora tenemos que declarar en el programa, más arriba, estos tipos que nos hemos inventado:

```
       TipoValor = (As, Dos, Tres, Cuatro, Cinco, Seis,
                    Siete, Sota, Caballo, Rey);
       TipoPalo = (Bastos, Oros, Copas, 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 para la carta.

```
1  procedure leercarta(ref carta: TipoCarta)
2      valor: TipoValor;
3      palo: TipoPalo;
4  {
5      write("valor: ");
6      read(valor);
7      write("palo: ");
8      read(palo);
9      carta = nuevacarta(valor, palo);
10 }
```

Resulta que tanto *valor* como *palo* son de tipos enumerados, por lo tanto podemos usar *read* para leerlos de la entrada. 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 un agregado para darle valor a la carta

```
carta = TipoPalo(valor, palo);
```

pero es mejor ocultar los detalles de una carta, y delegar la forma de crear una carta a un subprograma. Esta costumbre de utilizar funciones (o procedimientos) para crear o inicializar nuevos elementos del tipo resultará muy útil para tipos de datos más complicados. Esta función es simplemente:

```
1  function nuevacarta(valor: TipoValor, palo: TipoPalo): TipoCarta
2      carta: TipoCarta;
3  {
4      carta.valor = valor;
5      carta.palo = palo;
6      return carta;
7  }
```

¿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...

```
1  function valorcarta(carta: TipoCarta): float
2      pos: int;
3      valor: int;
4  {
5      if(esfigura(carta)){
6             return 0.5;
7      }else{
8             return 0;/* XXX esto falta XXX */
9      }
10 }
```

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 algunos escultores afirmen que lo mismo sucede con las esculturas, que estaban ya dentro de la piedra y que ellos sólo tenían que retirar lo que sobraba.

Nos falta escribir una parte de la función, como indica el comentario. ¿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. Podemos escribir:

```
1  function valorcarta(carta: TipoCarta): float
2      pos: int;
3      valor: int;
4  {
5      if(esfigura(carta)){
6             return 0.5;
7      }else{
8             pos = int(carta.valor);
9             valor = pos - int(As) + 1;
10            return float(valor);
11     }
12 }
```

¡No hemos dudado en declarar dos variables locales, *valor* y *pos*! Las hemos declarado con el único propósito de escribir la expresión

```
float(int(carta.valor) - int(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:

```
1  function esfigura(carta: TipoCarta): bool
2  {
3      return carta.valor >= Sota and carta.valor <= Rey;
4  }
```

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.

cartas.p

```
1     /*
2      *    Ver el valor de una carta en las 7 y 1/2
3      */

5     program cartas;

7     types:
8          TipoValor = (As, Dos, Tres, Cuatro, Cinco, Seis, Siete, Sota, Caballo, Rey);
9          TipoPalo = (Bastos, Oros, Copas, Espadas);
10         TipoCarta = record
11         {
12              valor: TipoValor;
13              palo: TipoPalo;
14         };

16    function nuevacarta(valor: TipoValor, palo: TipoPalo): TipoCarta
17         carta: TipoCarta;
18    {
19         carta.valor = valor;
20         carta.palo = palo;
21         return carta;
22    }

24    function esfigura(carta: TipoCarta): bool
25    {
26         return carta.valor >= Sota and carta.valor <= Rey;
27    }

29    function valorcarta(carta: TipoCarta): float
30         pos: int;
31         valor: int;
32    {
33         if(esfigura(carta)){
34              return 0.5;
35         }else{
36              pos = int(carta.valor);
37              valor = pos - int(As) + 1;
38              return float(valor);
39         }
40    }

42    procedure leercarta(ref carta: TipoCarta)
43         valor: TipoValor;
44         palo: TipoPalo;
45    {
46         write("valor: ");
47         read(valor);
48         write("palo: ");
49         read(palo);
50         carta = nuevacarta(valor, palo);
51    }
```

```
53    procedure main()
54          carta: TipoCarta;
55          valor: float;
56    {
57          leercarta(carta);
58          valor = valorcarta(carta);
59          write("La carta vale ");
60          writeln(valor);
61    }

—
```

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

```
types:
       TipoCarta = int;
```

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.

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 en 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 Picky 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 (sentencias entre ‘‘`{`’’ y ‘‘`}`’’) y la bifurcación condicional (todas las variantes de la sentencia *if* 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 estructuras de control 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){     /* mientras ... repetir ... */
        sentencias;
}                      /* fin del bucle */
```

Donde *inicialización* es una o varias sentencias utilizadas para preparar el comienzo del bucle propiamente dicho, que es el código desde *while* hasta ‘‘`}`’’. A ‘‘*while(condicion)*’’ se le llama normalmente **cabecera del bucle**. La cabecera del bucle determina si se ejecuta o no el cuerpo del bucle. Dicho cuerpo es el bloque de sentencias que sigue a la cabecera: desde ‘‘`{`’’ hasta ‘‘`}`’’. En muchas ocasiones no es preciso preparar nada para el bucle y la inicialización se puede omitir. La *condición* determina si se ejecuta el cuerpo del bucle o no (las sentencias entre las llaves). En el caso del bucle *while*, el bucle se repite mientras se cumpla la condición, evaluando siempre 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.

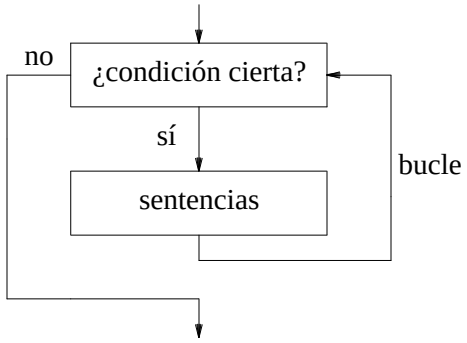


Figura 7.1: *Flujo de control en un bucle while.*

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, a pesar de que este término se considera muy coloquial (como podría resultar ‘‘tronco!’’ para referirse a una persona).

Veamos ahora 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){
       leercarta(carta);
       puntos = puntos + valorcarta(carta);
}
```

Cuando el programa llega a la cabecera del *while* (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 bucle y se ejecuta el cuerpo del bucle. Una vez ejecutadas las sentencias que hay dentro del bucle 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 del cuerpo del bucle. Pero si la condición es cierta se vuelve a repetir el bucle de nuevo.

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? Si nunca se sale del bucle, decimos que el bucle es un **bucle infinito**. En la mayoría de los casos un bucle infinito es un error. Aunque en algunos casos es justo lo que se quiere.

En el ejemplo, la primera vez entramos dado que *puntos* está inicializado a 0.0, 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 algo, 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 entraremos más al bucle. 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){
       leercarta(carta);
       puntos = puntos + valorcarta(carta);
}
if(puntos == 7.5){
       write("Has ganado");
}else{
       write("Has perdido");
}
writeeol();
```

7.2. Contar

El siguiente programa escribe los números del 1 al 5:

```
numero = 1;
while(numero <= 5){
	write(numero);
	numero = numero + 1;
}
```

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. Decir que se entra al bucle es lo mismo que decir que se ejecutan las sentencias del cuerpo del bucle: se imprime el número y se incrementa el número. Se sigue iterando (repitiendo, o dando vueltas) en el *while* mientras la condición sea cierta. ¡Por eso se llama *while*!

Se termina el bucle cuando el número pasa a valer 6, que no es menor o igual que 5. Fíjate que en la vuelta en la que el número pasa a valer 6, se escribe un 5 (ya que se escribe el número antes de incrementarlo). Después de esa iteración, se evalúa la condición, y como 6 no es menor o igual que 5, no se ejecuta el cuerpo del bucle y se continúa con las sentencias que hay después del bucle.

Es importante comprobar todo esto. Si el bucle hubiese sido

```
numero = 1;
while(numero < 5){
	write(numero);
	numero = numero + 1;
}
```

no habríamos escrito el número 5, dado que incrementamos número al final del bucle y justo después comprobamos la condición. En este caso, en la última iteración se escribe el número 4.

Si hubiésemos programado en cambio

```
numero = 1;
while(numero <= 5){
	numero = numero + 1;
	write(numero);
}
```

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.

Hay otro tipo de bucle que comprueba la condición al final de cada ejecución del cuerpo, no al principio. Este bucle se suele conocer como *do-while* (en otros lenguajes encontramos una variante llamada *repeat-until*) y lo podemos escribir en Picky como sigue:

```
do{
	sentencias;
}while(condición);
```

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 falsa, se termina la ejecución del bucle y se continúa tras él. Si esta es cierta, se sigue iterando.

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.

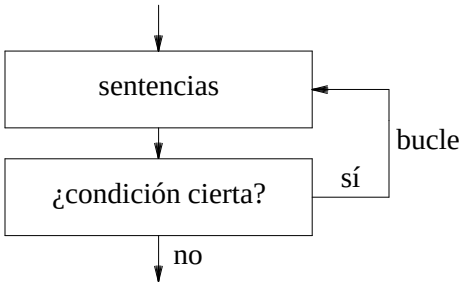


Figura 7.2: *Flujo de control en un bucle tipo do-while.*

```
numero = 1;
do{
       write(numero);
       numero = numero + 1;
}while(numero != 6);
```

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**. Aunque no habría sido así si la condición del *do-while* hubiera utilizado ‘‘<’’ en lugar de ‘‘!=’’. ¿Ves por qué?

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).

7.3. ¡Sabemos cuántas pasadas queremos, tronco!

Los dos tipos de bucle que hemos visto iteran un número variable de veces; siempre dependiendo de la condición: el bucle *while* comprobándola antes del cuerpo y el *do-while* comprobándola después.

Hay un tercer tipo de bucle que itera un número exacto de veces: ni una mas, ni una menos. Este bucle es muy popular dado que en muchas ocasiones sí sabemos el número de veces que queremos iterar. Nos referimos al bucle **for**.

El bucle *for* utiliza una variable, llamada **variable de control** del bucle, e itera siempre desde un valor inicial hasta un valor final. Por ejemplo, podría iterar de 1 a 5, de 0 a 4, etc. También puede iterar contando hacia atrás: de 5 a 1, de 10 a 4, etc. Dicho de otra forma, el bucle *for* itera haciendo que una variable tome los valores comprendidos en un rango.

La estructura del bucle *for* es como sigue:

```
for(inicialización , comparación){
       sentencias;
}
```

Igual que sucedía con el *while*, la parte *for(...)* es la *cabecera* del bucle y el bloque de sentencias que sigue es el cuerpo del bucle.

La *inicialización* de la cabecera del *for* da un valor inicial a la variable de control del bucle. Por ejemplo,

```
for(i = 1, ...){
       ...
}
```

hace que *i* sea la variable de control del bucle y que empiece por el valor `1`. Dicho de otro modo, en la primera iteración, la variable *i* valdrá `1`.

Tras la inicialización hay siempre una ‘‘`,`’’ y luego una comparación. La comparación determina cuál es el último valor para el rango de valores sobre el que iteramos. Dicho de otro modo, la comparación indica cuál será el último valor de la variable de control. Y dicho de otro modo aún, la comparación indica cuántas iteraciones haremos.

Se verá claro con un ejemplo. Este bucle itera desde 1 hasta 5:

```
for(i = 1, i <= 5){
       ...
}
```

Se suele leer este código diciendo: ‘‘desde *i* igual a 1 hasta 5, ...’’. Este bucle *for* ejecuta el cuerpo del bucle cinco veces. En la primera iteración *i* vale 1, en la segunda 2, y así hasta la última iteración, en que *i* vale 5.

También podemos escribir un bucle utilizando ‘‘`<`’’ en lugar de ‘‘`<=`’’. Por ejemplo:

```
for(i = 1, i < 5){
       ...
}
```

En este caso, el bucle va a ejecutar cuatro veces. En la última iteración, *i* valdrá 4. Vamos, iteramos sobre los valores 1, 2, 3 y 4. Este tipo de bucle *for* es muy habitual cuando sabemos sobre cuantos valores iteramos pero empezamos a contar en cero:

```
for(i = 0, i < NumManzanas){
       comermanzana(i)
}
```

Este otro bucle itera también 5 veces, pero cuenta desde 5 hasta 1:

```
for(i = 5, i >= 1){
       ...
}
```

Y este otro cuenta 4 veces, empieza a contar en 5 y pero termina en 2:

```
for(i = 5, i > 1){
       ...
}
```

Podrías pensar que la comparación de un *for* es una condición para seguir iterando, similar a un *while*. ¡Pero la comparación hace algo más! El hecho de que sea un ‘‘`<=`’’ o un ‘‘`<`’’ en la cabecera de un *for* hace que la variable se incremente automáticamente tras cada pasada del bucle, tras ejecutar el cuerpo del mismo. Si la condición es ‘‘`>`’’ o ‘‘`>=`’’, la variable se decrementa en lugar de incrementarse.

Por decirlo de otro modo. Se suele decir que *la variable de control itera en un rango de valores*. Dicho rango está definido por la inicialización (el primer elemento del rango) y por la comparación (que indica el último elemento del rango).

La variable de control tiene que estar declarada antes del bucle, y no es preciso que sea un *int* como en los ejemplos. Puede ser de cualquier tipo de datos **ordinal** (un tipo que se pueda contar). Por ejemplo, podemos utilizar: enteros, caracteres, booleanos o cualquier otro tipo enumerado. Los bucles *for* ascendentes (de menor a mayor) hacen que la variable *i* pase a valer *succ(i)* tras cada iteración. Los descendentes hacen que la variable *i* pase a valer *pred(i)* tras cada

iteración. Por ejemplo, este bucle imprime los caracteres que son letras minúsculas ('a'..'z'):

```
for(c = 'a', c <= 'z'){
      write(c);
}
```

Este otro bucle imprime los números del 1 al 9:

```
for(i = 1, i < 10){
      write(i);
}
```

Este otro imprime los enteros de 2 a 10 contados de 2 en 2:

```
for(i = 1, i <= 5){
      write(2*i);
}
```

Este otro bucle escribe la cuenta atrás del 10 al 0:

```
for(i = 10, i >= 0){
      write(i);
}
```

Puede que no iteremos ninguna vez. Eso pasa en aquellas ocasiones en que no se cumple la condición de comparación al comienzo del bucle. Esto puede suceder facilmente cuando comparamos la variable de control con otra variable en la condición del bucle *for*.

Volvamos a ver nuestro ejemplo de antes. El siguiente bucle es el que deberíamos escribir para imprimir los números del 1 al 5:

```
for(i = 1, i <= 5){
      write(i);
}
```

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. Lo repetimos para que veas que este bucle es exactamente igual a este otro:

```
i = 1;
while(i <= 5){
      write(i);
      i = i + 1;    /* i = succ(i) */
}
```

Pero el bucle *for* es más sencillo. 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 en el bucle la primera vez, en cuyo caso utilizamos un bucle *do-while*.

En muchas ocasiones la variable de control de un bucle *for* no se utiliza para nada mas 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 valor de la variable cada vez. En tal caso, para hacer algo 5 veces, escribiríamos un bucle *for* como el de arriba, pero sustituyendo la invocación a *write* por las sentencias que queramos ejecutar 5 veces.

La variable de control no puede modificarse en el cuerpo del bucle *for*, ya que usamos un bucle de este tipo para poder iterar en un rango de valores definido en la cabecera del bucle, osea, para ejecutar una acción un número conocido de veces. El propio bucle incrementa o decrementa la variable de control para conseguir este comportamiento. Muchos lenguajes prohiben la asignación de valores a la variable de control dentro del cuerpo del bucle. ¿Qué hará Picky?

Hay que tener en cuenta que 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.

Una última nota respecto a estos bucles. Es tradicional utilizar nombres de variable tales como *i*, *j* y *k* para controlar los bucles que iteran sobre enteros. 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.

7.4. 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.p

```
1     /*
2      *    Imprimir cuadrados de 1..5.
3      */

5     program cuadrados;

7     consts:
8           MaxNum = 5;

10    function cuadrado(n: int): int
11    {
12          return n ** 2;
13    }

15    procedure main()
16          x: int;
17          i: int;
18    {
19          for(i = 1, i <= MaxNum){
20                x = cuadrado(i);
21                write(i);
22                write(" ** 2 = ");
23                writeln(x);
24          }
25    }
```

—

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. Cuanto más podamos abstraer, mejor.

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. Para simplificar el código, supongamos que hemos definido un procedimiento para escribir el resultado llamado

escribecuadrado al que le pasamos el número y su cuadrado. Luego podríamos escribir

```
n = 1;
n2 = 1;
while(n2 <= 100){
       escribecuadrado(n, n2);
       n = n + 1;
       n2 = n ** 2;
}
```

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.p

```
1     /*
2      *    Imprimir cuadrados no mayores que 100.
3      */

5     program cuadrados;

7     procedure escribecuadrado(n: int, n2: int)
8     {
9           write(n);
10          write(" ** 2 = ");
11          writeln(n2);
12          writeeol();
13    }

15    procedure main()
16          n: int;
17          n2: int;
18    {
19          n = 1;
20          n2 = 1;
21          while(n2 <= 100){
22                escribecuadrado(n, n2);
23                n = n + 1;
24                n2 = n ** 2;
25          }
26    }
```

Podríamos haber escrito el bucle de este otro modo:

```
n = 1;
do{
       n2 = n ** 2;
       if(n2  <= 100){
              escribecuadrado(n, n2);
       }
       n = n + 1;
}while(n2 <= 100);
```

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 *do-while*. Debido a esto, en general, son más populares los bucles *while*.

7.5. 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(i = 1, i <= 10){
       for(j = 1, j <= 10){
              write("[");
              write(i);
              write(",");
              write(j);
              write("]");
              if(j < 10){
                     write(" ");
              }
       }
       writeeol();
}
```

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 *writeeol* tras el bucle más anidado?

7.6. 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.

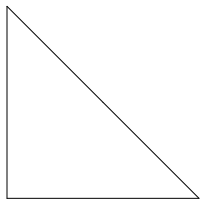


Figura 7.3: *Un triángulo rectángulo sobre un cateto.*

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):

```
; pick triangulo.p
; out.pam
*
**
***
****
*****
```

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 = 1, i <= Alto){
      ...
}
```

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.p

```
1     /*
2      *    Dibujar un triangulo rectangulo sobre un cateto.
3      */

5     program triangulo;

7     consts:
8          Alto = 5;

10    procedure main()
11         i: int;
12         j: int;
13    {
14         for( i = 1, i <= Alto){
15              for(j = 1, j <= i){
16                   write("*");
17              }
18              writeeol();
19         }
20    }
```

Ahora queremos dibujar dicho triángulo boca-abajo, como en la figura 7.4.

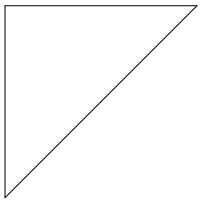


Figura 7.4: *Un triángulo invertido.*

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 = Alto, i > 0){
      for( j = 1, j <= i){
            write("*");
      }
      writeeol();
}
```

El resultado de ejecutar este programa es como sigue:

```
; pick trianguloinv.p
; out.pam
*****
****
***
**
*
```

Podríamos también centrar el triángulo sobre la hipotenusa, como muestra la figura 7.5. Este problema requiere pensar un poco más.

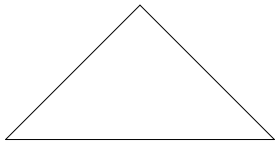


Figura 7.5: *Un triángulo sobre su hipotenusa.*

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 = 1, i <= Alto){
      ...
}
```

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 = 1, i <= Alto){
      /*
       * espacios a la izquierda.
       */
      nblancos = ?????
      for(j = 1, j <= nblancos){
            write(" ");
      }

      /*
       * relleno
       */
      ancho = ?????
      for(j = 1, j <= ancho){
            write("*");
      }

      /*
       * espacios a la derecha
       * no hacen falta.
       */
      writeeol();
}
```

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:

triangulobase.p

```
1     /*
2      *    Dibujar un triangulo rectangulo sobre un lado.
3      */

5     program triangulobase;

7     consts:
8          Alto = 5;

10    procedure main()
11         nblancos: int;
12         ancho: int;
13         i: int;
14         j: int;
15    {
16         for (i = 1, i <= Alto){
17              /* espacios a la izquierda. */
18              nblancos = Alto - i;
19              for(j = 1, j <= nblancos){
20                   write(" ");
21              }
22
23              /*
24               * relleno
25               */
26              ancho = 2 * i - 1;
27              for(j = 1, j <= ancho){
28                   write("*");
29              }

31              /*
32               * espacios a la derecha
33               * no hacen falta
34               */
35              writeeol();
36         }
37    }
```

Ejecutar el programa produce la siguiente salida:

```
; pick triangulobase.p
; out.pam
    *
   ***
  *****
 *******
*********
```

7.7. 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){
	if(esprimo(num)){
		writeln(num);
		llevamos = llevamos + 1;
	}
	num = num + 1;
}
```

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: int): bool
	loes: bool;
	i: int;
{
	loes = True;
	for(i = 2, i < num){
		if(num % i == 0){
			loes = False;
		}
	}
	return loes;
}
```

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.p

```
1	/*
2	 *	Imprime los primeros n primos
3	 */

5	program primos;

7	consts:
8		MaxPrimos = 15;
```

```
10    function esprimo(num: int): bool
11          loes: bool;
12          i: int;
13    {
14          loes = True;
15          i = 2;
16          while(i < num and loes){
17                loes = num % i != 0;
18                i = i + 1;
19          }
20          return loes;
21    }

23    procedure main()
24          num: int;
25          llevamos: int;
26    {
27          num = 1;
28          llevamos = 0;
29          while(llevamos < MaxPrimos){
30                if(esprimo(num)){
31                      writeln(num);
32                      llevamos = llevamos + 1;
33                }
34                num = num + 1;
35          }
36    }
—
```

Fíjate que también se ha eliminado el *if* de la función *esprimo*, ya que podemos asignar directamente el valor a la variable *loes*.

La salida del programa queda un poco fea, puesto que se imprime uno por línea. Podemos mejorar un poco el programa haciendo que cada cinco números primos se salte a la siguiente línea, y en otro caso separe el número con un espacio. Esto lo podemos hacer si modificamos el cuerpo del programa principal como sigue:

```
num = 1;
llevamos = 0;
while(llevamos < MaxPrimos){
      if(esprimo(num)){
            write(num);
            llevamos = llevamos + 1;
            if(llevamos % 5 == 0){
                  writeeol();
            }else{
                  write(" ");
            }
      }
      num = num + 1;
}
```

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?.

7.8. ¿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 Picky.

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 Picky.

20 Escribir los factores de un número.

21 Dibuja la gráfica de una función expresada en Picky 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 (por ejemplo, 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 $\vec{a} = (a_0, a_1, a_2)$ es un objeto compuesto de tres elementos del mismo tipo: a_0, a_1 y a_2. Además, a cada elemento le corresponde un número o índice (0 al primero, 1 al segundo y 2 al tercero). El concepto de *array* es similar y permite que, dado un índice, podamos recuperar un elemento de la colección de forma inmediata.

Podemos definir tipos *array* en Picky tal y como sigue:

```
types:
       TipoArray = 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 0 como primer elemento del rango. Esta forma de numerar (desde el 0) es muy común en las matemáticas y facilita la aritmética; ¡las matemáticas necesitan el cero! En otros lenguajes la costumbre es comenzar en 1, pero no así en Picky ni en lenguajes tan importantes como C, C++, Java, etc.

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 0 a 3 dado que se corresponde con una semana dada. Estas notas, conjuntamente, serían un vector de notas que podemos definir en Picky como sigue:

```
types:
      TipoNotas = array[0..3] of float;
```

Esto declara el nuevo tipo de datos *TipoNotas* como un *array* de cuatro elementos de tipo *float* (de reales) que tiene cuatro elementos con índices 0, 1, 2 y 3. Una vez declarado el tipo podemos declarar variables de dicho tipo, como por ejemplo:

```
notas: TipoNotas;
```

Habitualmente definiríamos un tipo 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:

```
consts:
       NumNotas = 4;

types:
       TipoRangoNotas = int 0..NumNotas-1;
       TipoNotas = 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 ordinal (enteros y enumerados como, por ejemplo, caracteres, días de la semana, etc.).

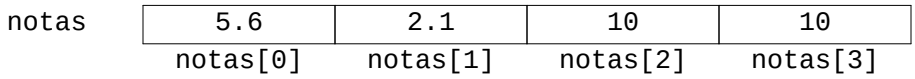


Figura 8.1: *Aspecto de un array en la memoria del ordenador.*

El siguiente tipo pretende describir cómo nos ha ido cada día de la semana:

```
types:
       TipoDiaSem = (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
       TipoQueTal = (MuyMal, Mal, Regular, Bien, MuyBien);
       TipoQueTalSem = 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 0 a 3 y que los elementos del *array* no son números reales, sino elementos de tipo *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 Picky escribimos el índice entre corchetes 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 índices usados para acceder a los elementos pueden elegirse en tiempo de ejecución, mientras el programa está ejecutando. Dicho de otra forma, la posición a la que se accede puede ser determinada por el valor de una variable. Esto significa que, dependiendo del estado del programa, se

puede acceder a un elemento o a otro. Esa es la principal ventaja de usar un *array* frente a usar una serie de variables. Por ejemplo, si *dia* es una variable de tipo *TipoDiaSemana*, podríamos ejecutar:

```
read(dia);
misemana[dia] = MuyBien;
```

En este ejemplo, la variable *dia* se está leyendo de la entrada, y cuando compilamos el programa no sabemos a que elemento del *array* se va a asignar el valor. Esto solo se sabrá cuando el programa esté ejecutando y se lea el valor de *dia*.

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 qué tal nos ha ido durante la semana:

```
for(dia = Lun, dia <= Dom){
       write(misemana[dia]);
}
```

Y cuidado aquí. *write* es un subprograma (un procedimiento) y los paréntesis se utilizan para representar una llamada (engloban el argumento de la llamada al procedimiento). En cambio, *misemana* es un *array* y los corchetes se utilizan para representar una indexación o acceso mediante índice al *array*. No debes confundirlos.

En Picky tenemos un operador **len** que nos da el número de elementos que contiene un tipo, variable, o constante. Podemos utilizarlo con un *array*. Así

```
len TipoQueTalSem
```

tiene como valor 7 (un *array* de ese tipo tiene siete elementos). Esto quiere decir que, por ejemplo, podemos recorrer un *array* indexado por enteros comenzando de 0 escribiendo:

```
for(i = 0, i < len miarray){
       ...
}
```

En este caso, *miarray* es una variable de un tipo *array*. Si alguna vez modificamos la longitud del tipo, no hará falta reescribir el bucle.

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, ya que el tipo *TipoNotas* tiene como rango 0..3:

```
valor = 77;
notas[valor] = 10;           /* 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*:

```
types:
       TipoVector = array[0..2] of int;

consts:
       Origen = TipoVector(0, 0, 0);
```

Igual que sucedía con los *record*, se permite asignar *arrays* del mismo tipo entre sí. Pero atención, no basta con que sean *arrays* del mismo tipo de elemento y con el mismo número de elementos: las variables tienen que ser del mismo tipo. La asignación de *arrays* copia los

elementos de uno, uno a uno, a los elementos de otro.

También se permite comparar *arrays* del mismo tipo entre sí, pero sólo comparaciones de igualdad o desigualdad. Estas operaciones funcionan elemento a elemento: la comparación considera que dos *arrays* son iguales si son iguales elemento a elemento.

Podemos querer tener vectores de varias dimensiones. En ese caso, en Picky debemos definir *arrays de arrays*. Por ejemplo, imaginemos que queremos un tipo de datos para manejar matrices de 4x3 elementos:

```
consts:
       NumFilas = 4;
       NumCols = 3;

types:
       TipoFila = array[0..NumCols-1] of float;
       TipoMatriz = array[0..NumFilas-1] of TipoFila;
```

Para inicializar una variable *matriz* de tipo *TipoMatriz* a ceros, podríamos hacer esto:

```
for(i = 0, i < NumFilas){
       for(j = 0, j < NumCols){
              matriz[i][j] = 0.0;
       }
}
```

Para imprimir la matriz, podríamos hacer lo siguiente:

```
for(i = 0, i < NumFilas){
       for(j = 0, j < NumCols){
              write(matriz[i][j]);
              if(j != NumCols-1){
                     write(", ");
              }
       }
       writeeol();
}
```

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*.

```
7     consts:
8           NumEstad = 5;

10    types:
11          TipoEstad = array[0..NumEstad-1] of int;
```

Utilizamos una constante *NumEstad* para poder cambiar fácilmente el tamaño de nuestra colección de estadísticas. De hecho, podríamos haber definido un 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:

```
16    function suma(estad: TipoEstad): int
17          sum: int;
18          i: int;
19    {
20          sum = 0;
21          for(i = 0, i < len estad){
22                sum = sum + estad[i];
23          }
24          return sum;
25    }
```

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 del *array*, en este caso del 0 al 3, ya que la condición del *for* es cierta mientras que *i* es menor que la longitud del *array*. En cada iteración, el bucle acumula (en la variable *sum*) el valor que hay en la posición *i* del *array*. El valor que tenemos acumulado en *sum* tras el bucle es el resultado buscado.

Para calcular la media, basta dividir la suma de todos los valores por el número de elementos. El programa que sigue incluye la función *media* y además ejercita el código que hemos escrito haciendo varias pruebas.

estad.p

```
1     /*
2      *    Imprimir estad.
3      */
```

```
5     program estad;

7     consts:
8           NumEstad = 5;

10    types:
11          TipoEstad = array[0..NumEstad-1] of int;

13    consts:
14          EstadPrueba = TipoEstad(1, 45, 3, 2, 4);   /* prueba */

16    function suma(estad: TipoEstad): int
17          sum: int;
18          i: int;
19    {
20          sum = 0;
21          for(i = 0, i < len estad){
22                sum = sum + estad[i];
23          }
24          return sum;
25    }

27    function media(estad: TipoEstad): int
28    {
29          return suma(estad) / len estad;
30    }

32    procedure main()
33    {
34          writeln(suma(EstadPrueba));
35          writeln(media(EstadPrueba));
36    }
```

—

¿A ti también te sale 55 y 11 como resultado? Es habitual necesitar subprogramas como estos que hemos hecho. Se hacen todos de un modo similar.

8.4. Buscar ceros

Imagina que 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 esqueleto de programa :

```
1     buscarcero(arraydenumeros, pos, encontrado);
2     if(encontrado){
3           write("pos = ");
4           write(pos);
5     }else{
6           write("no hay ninguno");
7     }
8     writeeol();
```

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     /*
2      * Busca un cero e informa de su posicion si se ha encontrado.
3      */
4     procedure buscarcero(nums: TipoNums, ref i: int, ref encontrado: bool)
5     {
6          encontrado = False;
7          i = 0;
8          while(i < len nums and not encontrado){
9               if(nums[i] == 0){
10                   encontrado = True;
11              }else{
12                   i = i + 1;
13              }
14         }
15    }
```

Como ves, inicialmente decimos que no lo hemos encontrado, y empezamos a iterar. En cada pasada, si el número es cero decimos que lo hemos encontrado (con lo que dejaremos de iterar y además *i* tendrá como valor la posición donde lo hemos encontrado). Fíjate en que si el número es el que buscamos entonces no incrementamos *i*. Además, tenemos que tener cuidado de no salirnos de rango en el índice. Si el *array* se nos acaba hay que dejar de iterar (y *encontrado* seguirá a *False*).

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. Otra forma de verlo que pensar que estamos buscando algún ‘‘no-cero’’, y ya sabemos cómo buscar en un *array*. Mostramos ahora el programa completo, con pruebas, en lugar de sólo el procedimiento.

todoceros.p

```
1     /*
2      *    Averigua si todos los numeros son cero en un array.
3      */

5     program todoceros;

7     consts:
8          NumElems = 5;

10    types:
11         TipoNums = array[0..NumElems-1] of int;

13    consts:
14         NumsPrueba = TipoNums(1, 45, 0, 2, 0);
15         CerosPrueba = TipoNums(0, 0, 0, 0, 0);
```

```
17    function todoscero(nums: TipoNums): bool
18          soncero: bool;
19          i: int;
20    {
21          soncero = True;
22          i = 0;
23          while(i < len nums and soncero){
24                if(nums[i] != 0){
25                      soncero = False;
26                }else{
27                      i = i + 1;
28                }
29          }
30          return soncero;
31    }

33    procedure main()
34    {
35          write("No son todos cero: ");
36          writeln(todoscero(NumsPrueba));
37          write("Son todos cero: ");
38          writeln(todoscero(CerosPrueba));
39    }
```

—

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. Buscar el mínimo se hace del mismo modo, pero en cada iteración hay que comprobar si el nuevo número es menor que nuestro candidato a mínimo, y actualizar el candidato en tal caso.

Este procedimiento es un ejemplo y devuelve tanto el máximo como el mínimo:

```
1     procedure extremos(nums: TipoNums, ref min: int, ref max: int)
2          i: int;
3     {
4          min = nums[0];
5          max = nums[0];
6
7          for(i = 1, i < len nums){
8               if(nums[i] < min){
9                    min = nums[i];
10              }
11              if(nums[i] > max){
12                   max = nums[i];
13              }
14         }
15    }
```

Sería fácil modificarlo para que además devolviese los índices en que se encuentran 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. En esta situación tomamos el menor elemento del resto del *array* para dejarlo justo en la posición *i* (intercambiandolo por dicho elemento como muestra la figura).

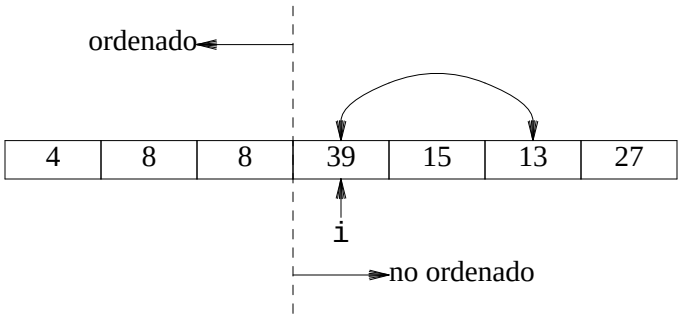


Figura 8.2: *Esquema para ordenar un array.*

Vamos a empezar ha hacer esto por el elemento cero del *array* y, en ese momento, no hay ninguna parte del *array* que esté ordenada. Podríamos empezar por programar algo como:

```
1     for(i = 0, i < len nums){
2           /*
2            *  buscar el menor de los que faltan por ordenar
3            *  cambiarlo por el de la posicion i
3            */
4     }
```

La pregunta es: ¿Cuáles son los que faltan por ordenar? Bueno, la idea de este bucle es que todos los que están en posiciones menores que *i* (inicialmente ninguno) ya están ordenados. Los que faltan son todos los números desde la posición *i* hasta el fin del *array* (inicialmente todos). Podríamos entonces escribir algo como esto:

```
1     for(i = 0, i < len nums){
2           buscarmenor(nums, i, posmin, min);
3           intercambiar(nums[i], nums[posmin]);
4     }
```

Utilizamos un procedimiento (que nos inventamos) *buscarmenor* al que pasamos como argumento el *array* que queremos ordenar, el índice desde donde nos falta por ordenar, una variable para que nos de el índice del menor elemento desde dicha posición (por referencia) y otra variable para que nos dé su valor (también por referencia). Podríamos haber escrito otro bucle *for* anidado para localizar el menor elemento desde el i-ésimo elemento hasta el último elemento, 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* no devuelve sólo su posición (*posmin*), también devuelve el menor valor (*min*).

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 previo de *nums[i]*. Y no queremos perderlo.

Aunque la idea está bien, tenemos que 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 *posmin* es igual que *i*; eso pasa cuando el elemento de la posición que estamos mirando (*i*) es el menor de los que restan por considerar. Entonces, podemos escribir:

```
1           for(i = 0, i < len nums - 1){
2                 buscarmenor(nums, i, posmin, min);
3                 if(posmin != i){
4                       intercambiar(nums[i], nums[posmin]);
5                 }
6           }
```

El programa completo podría quedar tal y como se ve a continuación. Este programa incluye un *array de arrays* para probar los procedimientos. En el programa principal se inicializa una variable con los *arrays* de prueba. Después en un bucle, se itera sobre el *array* de *arrays*, probando el subprograma de ordenación con todos los *arrays* de prueba. El programa escribe por su salida cada *array* antes y después de ser ordenado, dejando una línea en blanco entre cada prueba.

ordenar.p

```
1     /*
2      *    Ordenar una secuencia de numeros
3      */

5     program ordenar;
```

```
7     consts:
8          /*
9           * longitud de los arrays a ordenar
10          */
11         NumElems = 5;

13         /*
14          * numero de arrays de prueba
15          */
16         NumPruebas = 10;

18    types:
19         TipoIndice = int 0..NumElems-1;
20         TipoNums = array[TipoIndice] of int;

22         /*
23          *  tipos de datos para las pruebas
24          */
25         TipoIndicePruebas = int 0..NumPruebas-1;
26         TipoPruebas = array[TipoIndicePruebas] of TipoNums;

28    procedure escribirnums(ref nums: TipoNums)
29         i: int;
30    {
31         for(i = 0, i < len nums){
32              write(nums[i]);
33              if(i != len nums - 1){
34                   write(", ");
35              }
36         }
37         writeeol();
38    }

40    procedure intercambiar(ref n1: int, ref n2: int)
41         aux: int;
42    {
43         aux = n1;
44         n1 = n2;
45         n2 = aux;
46    }

48    procedure buscarmenor(nums: TipoNums, desde: int,  ref posmin: TipoIndice, ref min: int)
49         i: int;
50    {
51         min = nums[desde];
52         posmin = desde;
53         for(i = desde+1, i < len nums){
54              if(nums[i] < min){
55                   min = nums[i];
56                   posmin = i;
57              }
58         }
59    }
```

```
61    procedure ordenarnums(ref nums: TipoNums)
62         i: int;
63         min: int;
64         posmin: TipoIndice;
65    {
66         for(i = 0, i < len nums - 1){
67              buscarmenor(nums, i, posmin, min);
68              if(posmin != i){
69                   intercambiar(nums[i], nums[posmin]);
70              }
71         }
72    }

74    consts:
75         /* Constantes de prueba */
76         Pruebas = TipoPruebas(
77                   TipoNums(1, 45, 0, 2, 0),
78                   TipoNums(0, 0, -1, 0, 10),
79                   TipoNums(1, -4, -5, -6, -9),
80                   TipoNums(2, 3, 4, 5, 1),
81                   TipoNums(5, 3, 4, 1, 2),
82                   TipoNums(2, 1, 5, 4, 3),
83                   TipoNums(1, 5, 3, 2, 4),
84                   TipoNums(4, 1, 2, 3, 5),
85                   TipoNums(5, 2, 3, 4, 1),
86                   TipoNums(5, 1, 2, 3, 4)
87              );

89    procedure main()
90         i: int;
91         a: TipoNums;
92         pruebas: TipoPruebas;
93    {
94         /*
95          * inicializacion de los arrays de prueba
96          */
97         pruebas = Pruebas;
98         /*
99          * pruebas
100         */
101        for(i = 0, i < len Pruebas){
102             escribirnums(pruebas[i]);
103             ordenarnums(pruebas[i]);
104             escribirnums(pruebas[i]);
105             writeeol();
106        }
107   }
—
```

Ten en cuenta que como el programa modifica los *arrays* que ordena, no podemos ordenar directamente las constantes de prueba. Por eso hemos utilizado una variable.

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: TipoNums, valor: int, ref pos: int, ref esta: bool)
2          i: int;
3          puedeestar: bool;
4     {
5          i = 0;
6          puedeestar = True;
7          esta= False;

9          while(i < len nums and not esta and puedeestar){
10              esta= nums[i] == valor;
11              if(esta){
12                   pos = i;
13              }else{
14                   puedeestar = nums[i] < valor;
15                   i = i + 1;
16              }
17         }
18    }
```

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 *valor* no es menor que *nums[i]*, 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: TipoNums, valor: int, ref pos: int, ref esta: bool)
2          izq: int;
3          der: int;
4          med: int;
5          puedeestar: bool;
6     {
7          izq = 0;
8          der = len nums - 1;
9          esta = False;

11         while(not esta and izq <= der){
12              med = (izq + der) / 2;
13              if(valor == nums[med]){
14                   esta = True;
15                   pos = med;
16              }else if(valor > nums[med]){
17                   izq = med + 1;
18              }else{
19                   der = med - 1;
20              }
21         }
22    }
```

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 *izq* y *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).

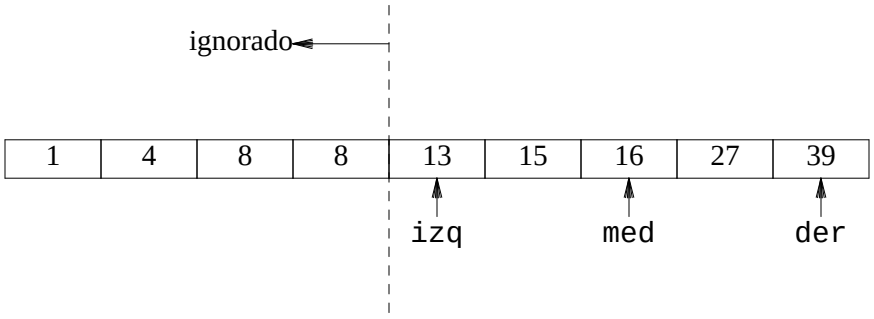


Figura 8.3: *Esquema de búsqueda binaria: partimos por la mitad en cada iteración.*

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**. Este tipo de datos se utiliza para manipular texto en los programas. Por eso es tan popular.

En Picky no tenemos un tipo de datos especial para las cadenas de caracteres. Una cadena de caracteres es un *array* que utiliza enteros como índice (comenzando en 0) y caracteres como elementos. Por ejemplo, un *string* para almacenar una palabra de 50 caracteres puede declararse como

```
consts:
       LongPalabra = 50;
types:
       TipoPalabra = array[0..LongPalabra-1] of char;
```

Para expresar constantes que sean cadenas de caracteres existe una sintaxis especial: se pueden escribir todos los caracteres del *array* entre comillas dobles, en lugar de usar una agregado. Si te fijas, llevamos usando literales de cadenas de caracteres como estos desde el principio del curso.

Por ejemplo, la siguiente declaración define una constante para un saludo:

```
consts:
       Saludo = "hola";
```

También podemos inicializar variables que sean *arrays* de caracteres de esta forma. Por ejemplo, si tenemos definido el tipo de datos:

```
consts:
       LongNombre = 4;
types:
       TipoNombre = array[0..LongNombre-1] of char;
```

podremos inicializar una variable *nombre* de tipo *TipoNombre* de esta forma:

```
       nombre = "Pepe";
```

Pero no podemos inicializar la variable con ‘‘`Marcos`’’ ni con ‘‘`Joe`’’ porque *TipoNombre* es un *array* de 4 caracteres, no de 5 ni de 3. En este ejemplo, en la memoria del ordenador tendremos un *array* de 4 caracteres como puede verse en la figura 8.4.

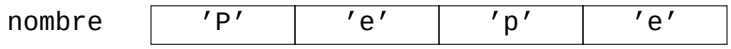


Figura 8.4: *Aspecto de una cadena de caracteres en la memoria del ordenador.*

Nótese que el tipo de datos al que pertenece el nombre anterior es distinto del tipo de datos *TipoPalabra* definido antes. 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: distinto tipo y longitud! */
```

La asignación anterior entre palabras es equivalente a ejecutar el siguiente bucle:

```
for(i = 0, i < len TipoPalabra){
       palabra1[i] = palabra2[i];
}
```

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). Lo mismo sucede con los *arrays* de caracteres (con los *strings*), naturalmente.

8.9. ¿Es un palíndromo?

Un palíndromo es una palabra que se lee igual al derecho que al revés. Queremos un programa que nos diga si una palabra es un palíndromo o no. Para el ejemplo, vamos a usar el tipo *TipoPalabra* que hemos definido en secciones anteriores. Nuestro problema puede definirse como la cabecera de función:

```
function espalindromo(s: TipoPalabra): bool
```

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, un palíndromo se lee igual al derecho que al revés. Si tenemos un procedimiento que invierte una cadena de caracteres, entonces podemos comparar la cadena original y la invertida para ver si son iguales. Luego podemos escribir...

```
1    function espalindromo(s: TipoPalabra): bool
2         sinvertido: TipoPalabra;
3    {
4         invertir(s, sinvertido);
5         return s == sinvertido;
6    }
```

Nos dan una variable de tipo *TipoPalabra* llamada *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 *TipoPalabra* que hay que invertir y devuelva en un segundo argumento el *TipoPalabra* ya invertido.

Pues está hecho. Si *s* es igual a *sinvertido* entonces *s* es un palíndromo.

Necesitamos ahora construir un *TipoPalabra* que sea la versión invertida de otro. Esto es un típico problema de construcción. La idea es que recorremos el *TipoPalabra* original y, para cada carácter, vamos a rellenar el carácter simétrico en el nuevo *TipoPalabra*, tal y como muestra la figura 8.5.

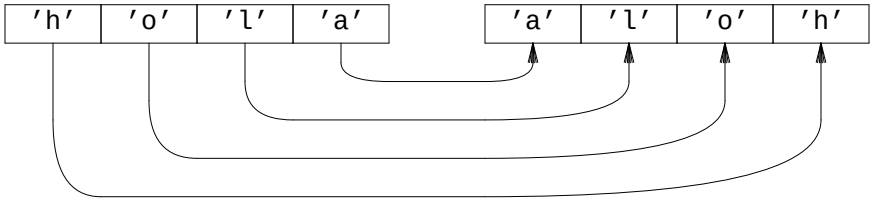


Figura 8.5: *Forma de invertir una cadena de caracteres.*

En el ejemplo de la figura habría que realizar las asignaciones siguientes:

```
1    sinvertido[3] = s[0];
2    sinvertido[2] = s[1];
3    sinvertido[1] = s[2];
4    sinvertido[0] = s[3];
```

Si recorremos los índices de *s*, desde 0, basta con asignarle *s[i]* a *sinvertido[4-1-i]*.

El siguiente programa incluye el procedimiento *invertir* junto con el resto del código y algún que otro caso de prueba.

palindromos.p

```
1    /*
2     *   Ver si una palabra es palindromo.
3     *   Version hecha top-down.
4     */

6    program palindromos;
```

```
8     consts:
9          LongPalabra = 4;
10         Prueba1 = "pepe";
11         Prueba2 = "otto";
12         Prueba3 = "sara";
13         Prueba4 = "abba";

15    types:
16         TipoRangoPalabra = int 0..LongPalabra-1;
17         TipoPalabra = array[TipoRangoPalabra] of char;

19    procedure invertir(s: TipoPalabra, ref inv: TipoPalabra)
20         i: int;
21    {
22         for(i = 0,  i < len TipoPalabra){
23              inv[len TipoPalabra - 1 - i] = s[i];
24         }
25    }

27    function espalindromo(s: TipoPalabra): bool
28         sinvertido: TipoPalabra;
29    {
30         invertir(s, sinvertido);
31         return s == sinvertido;
32    }

34    procedure main()
35    {
36         writeln(espalindromo(Prueba1));
37         writeln(espalindromo(Prueba2));
38         writeln(espalindromo(Prueba3));
39         writeln(espalindromo(Prueba4));
40    }
—
```

Este es el resultado de ejecutarlo:

```
; pick palindromos.p
; out.pam
False
True
False
True
```

En realidad este programa es muy ineficiente. Recorre el *array* 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 la cadena 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 espalindromo(s: TipoPalabra): bool
2          loes: bool;
3          i: int;
4     {
5          loes = True;
6          i = 0;
7          while(i < len s - 1 and loes){
8               loes = s[i] == s[len s - i - 1];
9               i = i + 1;
10         }
11         return loes;
12    }
```

Pero, ¡Un momento! No hace falta recorrer la cadena entera. 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 espalindromo(s: TipoPalabra): bool
2          med: int;
3          loes: bool;
4          i: int;
5     {
6          med = len s  / 2;
7          loes = True;
8          i = 0;
9          while(i < med and loes){
10              loes = s[i] == s[len s - i - 1];
11              i = i + 1;
12         }
13         return loes;
14    }
```

Ahora tenemos una función que sólo recorre la mitad de la cadena, lo que parece razonable para un programa eficiente.

8.10. 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     /*
2      * tipos para la baraja española
3      */
4     types:
5          TipoValor = (As, Dos, Tres, Cuatro, Cinco, Seis, Siete, Sota, Caballo, Rey);
6
7          TipoPalo = (Bastos, Oros, Copas, Espadas);
8
9          TipoCarta = record
10         {
11              valor: TipoValor;
12              palo: TipoPalo;
13         };
```

Si ahora queremos definir una mano podemos hacerlo sin más que definirla como un *array* de

cartas. Por ejemplo:

```
consts:
     NumCartas = 5;

types:
     ...

     TipoRangoMano = int 0..NumCartas-1;
     TipoMano = array[TipoRangoMano] of TipoCarta;
```

La definición de nuestro problema pasa a ser:

```
function valormano(mano: TipoMano): 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 programar directamente nuestro problema, siguiendo el ejemplo de la suma de números que vimos antes.

```
1    function valormano(mano: TipoMano): float
2         valor: float;
3         i: int;
4    {
5         valor = 0.0;
6         for(i = 0, i < NumCartas){
7              valor = valor + ValorCarta(mano[i]);
8         }
9         return valor;
10   }
```

¿Vés cómo el código es exactamente igual al del problema de sumar los números de un *array*? Es un problema de acumulación y se programa como todos los problemas de acumulación. Por eso es muy importante que domines todos los problemas que hemos visto: todo cuanto puedas querer programar termina siendo igual a uno de estos problemas.

8.11. Abstraer y abstraer hasta el problema demoler

Queremos ver qué caracteres están presentes en una cadena de caracteres arbitrariamente larga. ¿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 de la cadena 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 la cadena de caracteres, 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 nuestra cadena de caracteres de prueba, podríamos programar algo como esto:

```
program caracteres;

consts:
      LongCadena = 17;
      Prueba1 = "una muneca pelona";

types:
      TipoRangoCadena = int 0..LongCadena-1;
      TipoCadena = array[TipoRangoCadena] of char;
      ...

procedure main()
      c: TipoCjto;
      pos: int;
      elem: TipoElem;
{
      nuevocjto(c);
      for(pos = 0, pos < len Prueba1){
            elem = nuevoelem(Prueba1[pos]);
            insertarencjto(c, elem);
      }
      escribircjto(c);
}
```

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).

En el programa estamos usando un tipo *TipoElem* para abstraer el tipo de los elementos del conjunto. El centro del programa es el bucle *for*. Este bucle recorre toda nuestra cadena *Prueba1*, obteniendo el elemento correspondiente a cada carácter de la cadena mediante la funcion *nuevoelem*, para insertarlo después en el conjunto. ¿Insertar? Desde luego: nos hemos inventado un procedimiento *insertarencjto*, que inserta un elemento en un conjunto.

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 existe un número limitado y pequeño de caracteres distintos en la codificación de caracteres ASCII. 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.p

```
1    /*
2     *    Conjunto de caracteres.
3     */

5    program caracteres;

7    consts:
8         LongCadena = 17;
9         Prueba1 = "una muneca pelona";

11   types:
12        /*
13         * tipos para la cadena de prueba
14         */
15        TipoRangoCadena = int 0..LongCadena-1;
16        TipoCadena = array[TipoRangoCadena] of char;

18        /*
19         * tipos para el conjunto
20         */
21        TipoElem = char;
22        TipoCjto = array[TipoElem] of bool;

24   procedure nuevocjto(ref cjto: TipoCjto)
25        e: TipoElem;
26   {
27        for(e = Minchar, e <= Maxchar){
28             cjto[e] = False;
29        }
30   }

32   procedure insertarencjto(ref cjto: TipoCjto, elem: TipoElem)
33   {
34        cjto[elem] = True;
35   }

37   /*
38    * cjto es ref para evitar copia.
39    */
40   procedure buscarencjto(ref cjto: TipoCjto, elem: TipoElem, ref esta: bool)
41   {
42        esta = cjto[elem];
43   }
```

```
45    /*
46     * cjto es ref para evitar copia.
47     */
48    procedure escribircjto(ref cjto: TipoCjto)
49         e: TipoElem;
50    {
51         write('[');
52         for(e = Minchar , e <= Maxchar){
53              if(cjto[e]){
54                   write(e);
55              }
56         }
57         write(']');
58    }

60    function nuevoelem(c: char): TipoElem
61    {
62         return TipoElem(int(c));
63    }

65    procedure main()
66         c: TipoCjto;
67         pos: int;
68         elem: TipoElem;
69    {
70         nuevocjto(c);
71         for(pos = 0, pos < len Prueba1){
72              elem = nuevoelem(Prueba1[pos]);
73              insertarencjto(c, elem);
74         }
75         escribircjto(c);
76         writeeol();
77    }
—
```

Hemos definido el tipo *TipoElem* como un tipo *char*, para que el ejemplo sea mas completo. El *array* del tipo *TipoCjto* está indexado por dicho tipo enumerado. Podríamos haber utilizado directamente *char* como tipo de datos para los elementos y el código habría quedado algo más sencillo. No te preocupes mucho por esto.

Como nosotros hemos definido *TipoElem*, para poder insertar un carácter en el conjunto, necesitamos obtener su elemento correspondiente, para así poder indexar el *array* del conjunto. Si intentásemos pasar un *char* como segundo argumento para *insertarencjto*, no podríamos: el conjunto está indexado por objetos de tipo *TipoElem*, no por objetos de tipo *char*. Recuerda, no podemos mezclar tipos. Por eso necesitamos crear un elemento a partir del carácter. Para eso usamos la función *nuevoelem.* Esta función simplemente obtiene la posición del carácter, para después obtener el valor de la misma posición en el tipo *TipoElem*.

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:

```
; pick caracteres.p
; out.pam
[ 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, aunque sean de una línea. 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.

8.12. 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 usar un entero 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     consts:
2           NumMaxElems = 500;

6     types:

13          /*
14           * tipos para el conjunto
15           */
16          TipoElem = char;
17          TipoRangoElems = int 0..NumMaxElems-1;
18          TipoElems = array[TipoRangoElems] of TipoElem;
19          TipoCjto = record
20          {
21                elems: TipoElems;
22                nelems: int;
23          };
```

Para crear un conjunto vacío basta con que digamos que el número de elementos es cero.

```
28    procedure nuevocjto(ref cjto: TipoCjto)
39    {
30         cjto.nelems = 0;
31    }
```

Buscar un elemento requiere recorrerse todos los elementos que tenemos en *elems*, pero sólo los *nelems* primeros (dado que son los que tenemos en realidad). Por lo demás es nuestro procedimiento de búsqueda en una colección no ordenada.

```
34    procedure buscarencjto(ref cjto: TipoCjto, elem: TipoElem, ref esta: bool)
35         pos: int;
36    {
37         pos = 0;
38         esta = False;

40         while(pos < cjto.nelems and not esta){
41              if(cjto.elems[pos] == elem){
42                   esta = True;
43              }else{
44                   pos = pos + 1;
45              }
46         }
47    }
```

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!). Además, tenemos que tener en cuenta el tamaño del *array* en el que estamos guardando los elementos, que está determinado por la constane *NumMaxElems*. Sólo añadimos el nuevo elemento si no está en el conjunto y todavía caben más elementos. Añadirlo requiere asignar el elemento a la posición del *array* correspondiente, e incrementar el número de elementos.

```
49    procedure insertarencjto(ref cjto: TipoCjto, elem: TipoElem)
50         esta: bool;
51    {
52         buscarencjto(cjto, elem, esta);
53         if(not esta){
54              if(cjto.nelems == NumMaxElems){
55                   fatal("NumMaxElems ha de ser mayor!");
56              }else{
57                   cjto.elems[cjto.nelems] = elem;
58                   cjto.nelems = cjto.nelems + 1;
59              }
60         }
61    }
```

Fíjate cómo hemos utilizado *fatal* para imprimir un mensaje explicativo y abortar la ejecución del programa cuando una de las suposiciones que hemos hecho no se mantiene: cuando hay conjuntos de más de *NumMaxElems* elementos. Si tal cosa ocurre, sabremos qué ha ocurrido en lugar de tener que depurar el programa para averiguarlo.

Nos falta escribir los elementos del conjunto. Pero eso es fácil.

```
69    procedure escribircjto(ref cjto: TipoCjto)
70         pos: int;
71    {
72         write('[');
73         for(pos = 0 , pos < cjto.nelems){
74              write(cjto.elems[pos]);
75         }
76         write(']');
77         writeeol();
78    }
```

¡Terminado! Pero no sin mencionar antes una última cosa. El programa principal que utiliza este nuevo conjunto es como sigue:

```
80         nuevocjto(c);
81         for(pos = 0, pos < len Prueba1){
82              elem = nuevoelem(Prueba1[pos]);
83              insertarencjto(c, elem);
84         }
85         escribircjto(c);
```

¡Es el mismo programa principal de antes! Como hemos abstraído un conjunto de elementos 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.

8.13. ¡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]
```

¿Cómo pueden ser diferentes las salidas de ambos programas? 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 nuestra segunda versión del 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 la c justo en ese hueco.

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.

```
34    procedure buscarencjto(ref cjto: TipoCjto, elem: TipoElem, ref esta: bool, ref pos: int)
35         puedeestar: bool;
36    {
37         pos = 0;
38         esta = False;
39         puedeestar = True;

41         while(pos < cjto.nelems and not esta and puedeestar){
42              if(cjto.elems[pos] == elem){
43                   esta = True;
44              }else if(cjto.elems[pos] > elem){
45                   puedeestar = False;
46              }else{
47                   pos = pos + 1;
48              }
49         }
50    }
```

Ahora podemos modificar *insertarencjto*. La idea es que, como ya hemos dicho, si el elemento no está, 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.

```
52    procedure insertarencjto(ref cjto: TipoCjto, elem: TipoElem)
53         esta: bool;
54         i: int;
55         pos: int;
56    {
57         buscarencjto(cjto, elem, esta, pos);
58         if(not esta){
59              if(cjto.nelems > 0){
60                   for(i = cjto.nelems-1, i >= pos){
61                        cjto.elems[i+1] = cjto.elems[i];
62                   }
63              }
64              cjto.elems[pos] = elem;
65              cjto.nelems = cjto.nelems + 1;
66         }
67    }
```

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, consultar cuál es el elemento de la cima de la pila (el que está encima de todos los demás) y sacar un elemento de la cima de la pila (quitar el que está sobre todos los demás). 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 en el *array*.

14 Utilizar la pila del problema anterior para invertir una palabra. (Basta meter todos los caracteres de la palabra en una pila y luego extraerlos).

15 Calcular el valor de un número romano. Pista: dada una letra, su valor hay que sumarlo al valor total o restarlo del mismo dependiendo de si es mayor o igual al valor de la letra que sigue, o de si 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 caracteres al final, añadirlos al principio y añadirlos 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. Decimos que es persistente puesto que sigue ahí tras apagar el ordenador. Además de los ficheros ‘‘normales’’, hay que decir que tenemos otros: también son ficheros la entrada y la salida estándar (el teclado y la pantalla). 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. En este curso vamos a describir únicamente como manipular ficheros de texto, el resto de ficheros funciona de forma similar. Hasta ahora hemos estado usando la entrada y la salida estándar, que son ficheros. Lo seguiremos haciendo pero, a partir de ahora, utilizaremos los ficheros de un modo más controlado que hasta el momento.

En Picky los ficheros están representados por el tipo de datos `file`. Este tipo de datos sirve para representar dentro del lenguaje algo que en realidad está fuera de él, 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 llamado *README.first* para leer de él, asociándolo con la variable *fichdatos*, de tipo *file*.

```
open(fichdatos, "README.first", "r");
```

El primer argumento es el fichero tal y como lo representa Picky. El segundo argumento es el nombre del fichero tal y como lo conoce el sistema operativo. El tercer argumento indica si queremos leer o escribir en el fichero (en este caso la ‘‘`r`’’, de **read** en inglés, siginifica que se quiere leer). A partir de esta sentencia podemos utilizar la variable *file* para referirnos al fichero llamado *README.first* en el ordenador, para leer de él. 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 estándar. Picky incluye dos variables predefinidas, *stdin* y *stdout*, ambas de tipo *file*, que corresponden a la entrada y a la salida del programa.

Para guardar datos en un fichero tendremos que abrirlo para escribir y no para leer. Esto se hace utilizando una sentencia similar a la anterior, pero pasando como tercer argumento ‘‘`w`’’, de **write** en inglés, en lugar de ‘‘`r`’’. Cuando lo hacemos así, el fichero se vacía y se prepara para que escribamos datos en él (Si el fichero no existía, Picky lo crea).

Una vez hemos terminado de utilizar un fichero es preciso llamar al procedimiento *close* para **cerrar** el fichero. Por ejemplo:

```
close(fichdatos);
```

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. El procedimiento *close* es sólo para cerrar ficheros que abrimos nosotros.

Del mismo modo que hemos estado utilizando procedimientos *read*, *write*, *readln* y *writeln* para leer de la entrada y escribir en la salida estándar, podemos utilizar los procedimientos llamados *fread*, *fwrite*, *freadln* y *fwriteln* para leer de cualquier fichero y escribir en cualquier fichero. Estos subprogramas requieren pasar como primer argumento la variable de tipo *file* que representa al fichero que hemos abierto. Igual sucede con el procedimiento *writeeol*, cuya versión para un fichero abierto por nosotros se llama *fwriteeol*. Como ves, las funciones y procedimientos que estamos viendo en este tema tienen una versión que comienza por la letra ‘‘`f`’’ y que requiere pasar un argumento de tipo *file* para poder usar los ficheros que abrimos, y una versión que no empieza por ‘‘`f`’’ para tratar la salida y/o la entrada estándar.

Por ejemplo, este programa abre un fichero llamado *datos.txt* y reemplaza su contenido por una única línea que contiene el texto ‘‘`hola`’’:

escribehola.p

```
1     /*
2      *    Escribe un saludo en datos.txt
3      */

5     program escribehola;

7     procedure main()
8          fichdatos: file;
9     {
10         open(fichdatos, "datos.txt", "w");
11         fwriteln(fichdatos, "hola");
12         close(fichdatos);
13    }

—
```

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. También podríamos haber hecho esto:

```
open(fichdatos, "datos.txt", "w");
fwrite(fichdatos, "hola");
fwriteeol(fichdatos);
close(fichdatos);
```

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.p

```
1     /*
2      *    Lee un saludo en datos.txt
3      */

5     program leehola;

7     types:
8          TipoCadena  = array[0..3] of char;
```

```
10    procedure main()
11          fichdatos: file;
12          cadena: TipoCadena;
13    {
14          open(fichdatos, "datos.txt", "r");
15          fread(fichdatos, cadena);
16          close(fichdatos);
17          writeln(cadena);
18    }
—
```

Si creamos un fichero de texto que contenga ‘‘`hola`’’ al principio y ejecutamos nuestro programa podemos ver lo que pasa:

```
; pick leehola.p
; out.pam
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`’’.

```
; out.pam
p.p:15: read: eof met
pc=0x1a, sp=0x8c, fp=0x8c
```

¡El programa ha sufrido un error! Hemos intentado leer cuatro caracteres y en el fichero no había tantos. 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.

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. Dicha marca suele estar formada por uno o más caracteres y su valor concreto depende del sistema operativo que utilizamos. Si del fichero anterior leemos un carácter

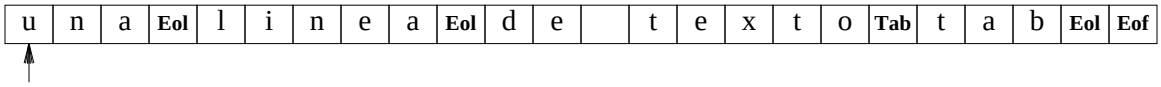


Figura 9.1: *Fichero del que lee el programa.*

entonces estaremos en la situación que ilustra la figura 9.2.

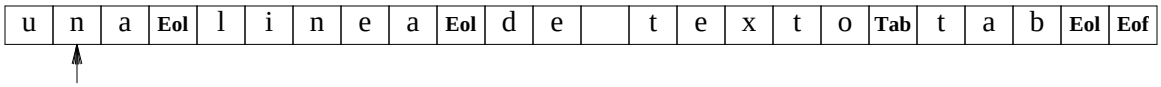


Figura 9.2: *Fichero tras leer un carácter.*

Se habrá leído el carácter ‘‘u’’ y la posición en el fichero habrá avanzado un carácter. Si ahora leemos dos caracteres más, pasaremos a la situación ilustrada en la figura 9.3.

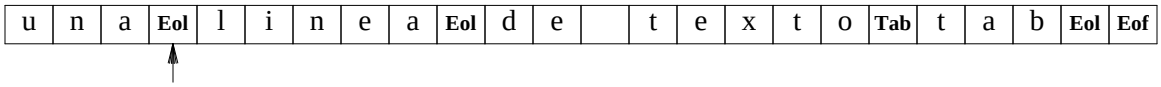


Figura 9.3: *Fichero tras haber leído dos caracteres más.*

En este momento la posición de lectura u *offset* de la entrada está apuntando a **Eol**, el **fin de línea**. Una vez aquí, si se intenta leer un carácter, no se lee la siguiente letra (la ‘‘l’’ en este ejemplo). La variable de tipo *char* que pasamos a *read* pasa a tener el valor de la constante predefinida *Eol*, para indicarnos que hemos llegado a una marca de fin de línea. No podemos volver a leer un carácter del fichero en este caso. Si lo hacemos, tendremos un error de ejecución.

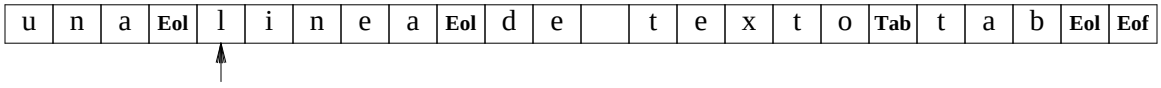
La marca de fin de línea tiene que tratarse de una forma especial. Esto es así porque la marca de fin de línea varía de un sistema operativo a otro. Esta marca es en realidad uno o más caracteres que indican que se salta a otra línea en el texto, y habitualmente la escribimos pulsando la tecla *Intro*.

Para saltar la marca de fin de línea debemos usar el procedimiento *freadeol* (o *readeol* si se trata de la entrada estándar). Siguiendo con nuestro ejemplo, al llamar a *freadeol* la posición del fichero salta el fin de línea y pasa a estar en el siguiente carácter (la ‘‘l’’).

¿Cómo sabemos si tenemos que saltar un fin de línea? Es fácil. Después de intentar leer un carácter del fichero, debemos usar una función llamada *feol* (o *eol* si se trata de la entrada estándar) para comprobar si al leer se llegó a un *Eol* o no. Dicha función devuelve *True* si se está en un fin de línea.

Por cierto, como no es un carácter, no podemos escribirlo normalmente. Para escribir un fin de línea, necesitamos usar el procedimiento *fwriteeol* (o *writeeol* si se trata de la salida estándar).

Seguimos con nuestro ejemplo. Una vez se ha saltado el fin de línea, nuestro 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*).

Igual que la marca de fin de línea, la marca de fin de fichero no es un carácter que se pueda leer. Si se intenta leer un carácter y se llega a la marca de fin de fichero, el carácter que hemos pasado como argumento pasa a tener el valor de la constante predefinida *Eof*. En ese caso, tenemos que dejar de leer el fichero. Si intentamos leer de nuevo, obtendremos un error de ejecución.

Para saber si estamos en una situación de fin de fichero, después de intentar leer, podemos llamar a la función *feof* (o *eof* si se trata de la entrada estándar). Esta función devuelve *True* cuando se está en el fin de fichero. En ese caso, deberemos cerrar el fichero, porque ya no podremos leer más caracteres de él. Pero recuerda que la entrada estándar nunca se debe cerrar, puesto que no la hemos abierto nosotros.

Por cierto, un fichero que abrimos para escribir, está vacío justo después de la llamada a *open* y se encuentra en una situación de fin de fichero. Leer de él 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*. Cuando escribimos un *Control-d* en Linux, el sistema operativo hace que la lectura de teclado se encuentre un fin de fichero.

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.p

```
1     /*
2      *    Escribe el contenido de datos.txt en su salida
3      */

5     program copiaentrada;

7     procedure main()
8          fichdatos: file;
9          c: char;
10    {
11         open(fichdatos, "datos.txt", "r");
12         do{
13              fread(fichdatos, c);
14              if(feol(fichdatos)){
15                   freadeol(fichdatos);
16                   writeeol();
17              }else if(not feof(fichdatos)){
18                   write(c);
19              }
20         }while(not feof(fichdatos));
21         close(fichdatos);
22    }
```

Nótese que el programa tiene extremo cuidado de no volver a llamar a *read* cuando se termina el fichero. Por eso está toda la lectura encerrada en un *do-while* que utiliza *not feof(fichdatos)* como condición.

Hay que tener en cuenta que la función *feof* nos indica si en la **última lectura** se alcanzó el final de fichero. No deberíamos plantear el bucle como sigue:

```
while(not feof(fichdatos)){
      ...
}
```

ya que estaríamos llamando a *feof* antes de haber leído del fichero. Lo mismo pasaría con *feol*, *eol*

y *eof*. Dichas funciones no leen del fichero (¡las funciones no deben tener efectos laterales!) para ver lo que viene, sino que nos dicen si nos hemos encontrado la marca en la última lectura que se ha realizado.

Otro detalle importante es que el programa no intenta siquiera leer los fines de línea presentes en el fichero. Así, si después de leer, *feol* indica que estamos mirando al fin de línea, el programa evita dicha marca de fin de línea llamando a *freadeol* y después escribe el fin de línea llamando a *writeeol*.

Por último, también tiene mucho cuidado con no escribir el carácter que se pasó a *fread* si se ha llegado al final de fichero. Si se llegó al fin del fichero, *c* tiene el valor de la constante que sirve para indicar la marca, pero no es un carácter que se pueda escribir. Si se intenta escribir dicho valor, habrá un error de ejecución.

9.3. Lectura controlada

Cuando leemos una cadena de caracteres, *fread* (y *read*) consume de la entrada tantos caracteres como tiene la cadena considerada. Si, por ejemplo, el usuario escribe menos caracteres en el teclado, entonces se producirá un error en tiempo de ejecución. Así, dadas las definiciones

```
consts:
       LongCadena = 10;

types:
       TipoCadena = array[0..LongCadena-1] of char;
```

si tenemos una variable *nombre* de tipo *TipoCadena*, la sentencia

```
fread(nombre);
```

leerá exactamente 10 caracteres, produciéndose un error si no tenemos dichos caracteres en la entrada (esto es, si se acaba el fichero antes). 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 *fread* (o *read* si es la entrada) para leer enteros o números reales, este procedimiento se salta los 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). Naturalmente, para saltarse los blancos, los lee. Si cuando llamamos a *read* (o *fread*) no hay un número en la entrada, 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 *read*. A tal efecto existe un procedimiento llamado *fpeek* (o *peek* si se trata de la entrada estándar).

A este procedimiento se le pasa un carácter por referencia. El procedimiento mira en el fichero (lo que internamente requiere leer del mismo) y nos devuelve el carácter que leeremos a continuación de dicho fichero. Si lo que viene a continuación es un fin de línea *fpeek* devuelve (en el carácter que le pasamos por referencia) la constante *Eol*. Si el fichero no tiene más datos, devuelve la constante *Eof*. En cualquier otro caso, devuelve el siguiente carácter, que se leerá en la próxima llamada para leer.

Por ejemplo, la sentencia

```
peek(c);
```

hace que *c* tenga el valor de la constante *Eol* si estamos a punto de leer un fin de línea de la entrada estándar. Esta otra sentencia hace lo mismo pero para el fichero *fichdatos* y no para la entrada estándar:

```
    fpeek(fichdatos, c);
```

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.p

```
1     /*
2      *    Lee entrada y se salta los blancos.
3      */

5     program sinblancos;

7     procedure main()
8          c: char;
9     {
10         do{
11              peek(c);
12              switch(c){
13              case Eol:
14                   readeol();
15                   writeeol();
16              case Eof:
17                   ;    /* no hay que hacer nada mas */
18              default:
19                   read(c);
20                   if(c != Tab and c != ' '){
21                        write(c);
22                   }
23              }
24         }while(not eof());
25    }
```

La idea es que el programa se mantendrá leyendo de la entrada estándar mientras no se haya producido un fin de fichero. Lo primero que hace el programa es llamar a *peek.* Dependiendo de lo que averigüe dicha función el programa hará una cosa u otra.

Por un lado, puede que *peek* diga que hay un fin de línea en la entrada. En tal caso *c* será *Eol* y el programa se salta el fin de línea llamando a *readeol()*, y lo escribe en la salida llamando a *writeeol()*.

Ahora bien, si *peek* ha averiguado que estamos ante un fin de fichero (esto es, si *c* vale *Eof*) entonces tenemos que dejar de iterar. De eso se va a ocupar la condición del *do-while*, por lo que no tenemos que hacer nada mas.

Si no tenemos ni un fin de línea ni un fin de fichero, tendremos un carácter que podemos procesar:

```
21                   read(c);
22                   if(c != Tab and c != ' '){
23                        write(c);
24                   }
```

¡Primero hay que leerlo! Es cierto que ya sabemos lo que es `c`. Nuestro espía, *peek*, nos lo ha dicho antes siquiera de que lo leamos. 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 *read* entonces *peek* volverá a decir lo mismo la siguiente vez que lo llamemos: crearemos un bucle infinito.

Después, como el programa debe escribir todo lo que lee salvo que sea un blanco, lo que tenemos que hacer es un *write* del carácter si este no es un espacio en blanco o un tabulador (utilizamos la constante *Tab*, que es el carácter de tabulación horizontal).

9.4. Separar palabras

Queremos leer controladamente la entrada para escribir a la salida una palabra por línea. Suponemos que las palabras están formadas por letras minúsculas sin acentuar (esto es, ‘‘`ab33cd`’’ se considera como dos palabras distintas: ‘‘`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     do{
2          saltarblancos();
3          if(not eof()){
4               leerpalabra(palabra);
5               escribirpalabra(palabra);
6               writeeol();
7          }
8     }while(not eof());
```

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.

Para empezar a programar esto poco a poco, vamos a simplificar. Podemos empezar por saltarnos los caracteres que forman una palabra en lugar de leer una palabra. Nuestro programa podría quedar así:

```
1     do{
2          saltarblancos();
3          if(not eof()){
4               saltarpalabra();
5          }
6     }while(not eof());
```

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()
2           c: char;
3           fin: bool;
4     {
5           fin = False;
6           do{
7                 peek(c);
8                 switch(c){
9                 case Eol:
10                      readeol();
11                case Eof:
12                      fin = True;
13                default:
14                      if(esblanco(c)){
15                            read(c);
16                      }else{
17                            fin = True;
18                      }
19                }
20          }while(not fin);
21    }
```

Tras llamar a *peek* tenemos que ver no sólo si tenemos un fin de línea o un fin de fichero; tenemos que ver si tenemos un blanco u otra cosa. Si es un blanco, hay que leerlo para que avance la posición del fichero. 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, no tenemos un bucle que utilice como antes la función *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. Si en algún momento se detecta el final del fichero, simplemente se pone *fin* a cierto para salir del bucle y retornar. El subprograma no requiere tener parámetros, ni siquiera para notificar al que le llama si se ha llegado al final del fichero; el llamador puede usar la función *eof* para saber si se alcanzó el fin del fichero. Si, tras llamar a *saltarblancos*, no estamos en fin de fichero entonces es seguro que tenemos una palabra en la entrada.

El procedimiento *saltarpalabra* es más sencillo, porque sabe que al menos tiene un carácter para la palabra en la entrada. Esto es así puesto que se supone que antes se ha llamado a *saltarblancos* y se ha comprobado *eof*, como hemos dicho. Podemos leer el carácter, y en función de lo que tengamos detrás, seguimos leyendo o no.

```
1     procedure saltarpalabra()
2           c: char;
3     {
4           do{
5                 read(c);
6                 write(c);
7                 peek(c);
8           }while(c != Eol and c != Eof and not esblanco(c));
9     }
```

Aquí, para poder ver si el programa hace lo que debe, o no, escribimos en la salida los caracteres que consideramos como parte de la palabra. Al menos podemos probar el programa y ver lo que pasa.

Y nos falta la función *esblanco*. Como ya comentamos, consideramos como ‘‘blanco’’, para este ejemplo, cualquier carácter que no sea una letra minúscula:

```
1     function esblanco(c: char): bool
2     {
3          return c < 'a' or c > 'z';
4     }
```

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 una cadena de caracteres (*string*) es lo más adecuado para almacenarla. La pregunta es: ¿De qué tamaño?

Podemos utilizar una cadena de un tamaño tal que nos baste para cualquier palabra que podamos encontrar y guardar, además, cuántos caracteres de dicha cadena son los que forman parte de la palabra. Empezamos por fijar el tamaño máximo de palabra que nuestro programa soportará:

```
1     consts:
2          MaxLongPalabra = 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 y cuántos son basura:

```
1     types:
2          TipoRangoPalabra = int 0..MaxLongPalabra-1;
3          TipoNatural = int 0..Maxint;
4          TipoLetras = array[TipoRangoPalabra] of char;
5          TipoPalabra = record
6          {
7               letras: TipoLetras;
8               long: TipoNatural;
9          };
```

Si tenemos una palabra *pal* entonces sabemos que los caracteres de la palabra son en realidad los caracteres comprendidos entre *pal.letras[0]* y *pal.letras[pal.long-1]*. También hemos definido un tipo nuevo para los números naturales (0, 1, 2...) llamado *TipoNatural*, para la longitud de la palabra.

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(ref palabra: TipoPalabra)
2          c: char;
3     {
4          palabra.long = 0;     /* palabra vacia por ahora */
5          do{
6               read(c);
7               palabra.letras[palabra.long] = c;
8               palabra.long = palabra.long + 1;
9               peek(c);
10         }while(c != Eol and c != Eof and not esblanco(c));
11    }
```

Tras la llamada a *read* insertamos el carácter en la posición correspondiente del array *letras*, y después incrementamos la longitud de la palabra. Dado que el array comienza en 0, *palabra.long*

es el índice donde debemos insertar. Después debemos incrementar la longitud, ya que hemos insertado un nuevo carácter. Hemos ignorado por completo qué pasa si hay palabras más largas en la entrada del programa.

Estamos cerca. Nos falta *escribirpalabra*. Pero este es muy fácil:

```
1     procedure escribirpalabra(palabra: TipoPalabra)
2           i: TipoNatural;
3     {
4           for(i = 0, i < palabra.long){
5                 write(palabra.letras[i]);
6           }
7     }
```

Bastaba llamar a *write* para cada carácter de *palabra.letras* que se está usando (que no excede de la longitud de la palabra). El programa completo quedaría como sigue:

leerpalabras.p

```
1     /*
2      *    Leer palabras separadas
3      */

5     program leerpalabras;

7     consts:
8           MaxLongPalabra = 500;

10    types:
11          TipoRangoPalabra = int 0..MaxLongPalabra-1;
12          TipoNatural = int 0..Maxint;
13          TipoLetras = array[TipoRangoPalabra] of char;
14          TipoPalabra = record
15          {
16                letras: TipoLetras;
17                long: TipoNatural;
18          };


21    function esblanco(c: char): bool
22    {
23          return c < 'a' or c > 'z';
24    }
```

```
26    procedure saltarblancos()
27         c: char;
28         fin: bool;
29    {
30         fin = False;
31         while(not fin){
32              peek(c);
33              switch(c){
34              case Eol:
35                   readeol();
36              case Eof:
37                   fin = True;
38              default:
39                   if(esblanco(c)){
40                        read(c);
41                   }else{
42                        fin = True;
43                   }
44              }
45         }
46    }

48    procedure leerpalabra(ref palabra: TipoPalabra)
49         c: char;
50    {
51         palabra.long = 0;    /* palabra vacia por ahora */
52         do{
53              read(c);
54              palabra.letras[palabra.long] = c;
55              palabra.long = palabra.long + 1;
56              peek(c);
57         }while(c != Eol and c != Eof and not esblanco(c));
58    }

60    procedure escribirpalabra(palabra: TipoPalabra)
61         i: TipoNatural;
62    {
63         for(i = 0, i < palabra.long){
64              write(palabra.letras[i]);
65         }
66    }

68    procedure main()
69         palabra: TipoPalabra;
70    {
71         do{
72              saltarblancos();
73              if(not eof()){
74                   leerpalabra(palabra);
75                   escribirpalabra(palabra);
76                   writeeol();
77              }
78         }while(not eof());
79    }
```

—

Este programa se puede mejorar. Por ejemplo, ¿Qué pasaría si nos encontramos una palabra de más de *MaxLongPalabra* caracteres? Intenta mejorar *leerpalabra* para evitar ese error.

9.5. La palabra más larga

Queremos imprimir, en la salida estándar, la palabra más larga presente en la entrada del programa. De haber programado el problema anterior en un sólo procedimiento, justo para escribir una palabra por línea, sería más complicado resolver este problema. Pero como lo hicimos bien, implementando subprogramas para cada problema que nos encontramos, ahora es muy fácil resolver otros problemas parecidos.

Veamos. Suponiendo de nuevo que tenemos todo lo que podamos desear, podríamos programar algo como lo que sigue:

```
1          maslarga = nuevapalabra();
2          do{
3               saltarblancos();
4               if(not eof()){
5                    leerpalabra(palabra);
6                    if(longpalabra(palabra) > longpalabra(maslarga)){
7                         maslarga = palabra;
8                    }
9               }
10         }while(not eof());
```

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 candidata a 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): TipoNatural
2     {
3          return palabra.long;
4     }

6     function nuevapalabra(): TipoPalabra
7          palabra: TipoPalabra;
8     {
9          palabra.long = 0;
10         return palabra;
11    }
```

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){
2               write("No hay palabras");
3          }else{
4               escribirpalabra(maslarga);
5          }
6          writeeol();
```

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.

9.6. ¿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.

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.

9.7. 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 Picky 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     TipoPositivo = int 1..Maxint;
2     TipoFrec = record
3     {
4          palabra: TipoPalabra;
5          veces: TipoPositivo;
6     };
```

El número de veces que aparece una palabra será un número positivo (entero mayor o igual que 1). Nos hemos definido un tipo *TipoPositivo* para ello.

Si ponemos un límite al número máximo de palabras que podemos leer (cosa necesaria en este problema) podemos declarar entonces la siguiente constante en su correspondiente sección:

```
1     MaxNumPalabras = 1000;
```

Ahora ya nos podemos definir el array en la sección de tipos:

```
1     TipoRangoPalabras = int 0..MaxNumPalabras-1;
2     TipoFrecs = 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     TipoCjtoFrecs = record
2     {
3          frecs: TipoFrecs;
4          numfrecs: TipoNatural;
5     };
```

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          cjto = nuevocjto();
2          do{
3               saltarblancos();
4               if(not eof()){
5                    leerpalabra(palabra);
6                    insertar(cjto, palabra);
7               }
8          }while(not eof());
```

Hemos vaciado nuestro conjunto de frecuencias (¡Que también nos hemos inventado!) con *nuevocjto* 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 *cjto.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(cjto, palabra);
2     escribirpalabra(palabra);
3     writeeol();
```

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(ref c: TipoCjtoFrecs, palabra: TipoPalabra)
2          pos: TipoNatural;
3          encontrado: bool;
4     {
5          buscar(c, palabra, encontrado, pos);
6          if(encontrado){
7               incrfrec(c.frecs[pos]);
8          }else{
9               c.numfrecs = c.numfrecs + 1;
10              c.frecs[pos] = nuevafrec(palabra);
11         }
12    }
```

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 dos palabras. Para ello, debe comprobar 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. Este procedimiento utiliza el valor 0 para *maxveces* para detectar dicho caso (si hay algún elemento en el conjunto, la variable *maxveces* debe ser mayor que 0 después del bucle *for*).

El programa completo quedaría como sigue:

masfrecuente.p

```
1     /*
2      *    Lee palabras y escribe la mas frecuente
3      */

5     program masfrecuente;

7     consts:
8          MaxLongPalabra = 500;
9          MaxNumPalabras = 1000;

11    types:
12         TipoNatural = int 0..Maxint;
13         TipoPositivo = int 1..Maxint;

15         TipoRangoPalabra = int 0..MaxLongPalabra-1;
16         TipoRangoPalabras = int 0..MaxNumPalabras-1;

18         TipoLetras = array[TipoRangoPalabra] of char;
19         TipoPalabra = record
20         {
21              letras: TipoLetras;
22              long: TipoNatural;
23         };

25         TipoFrec = record
26         {
27              palabra: TipoPalabra;
28              veces: TipoPositivo;
29         };
30         TipoFrecs = array[TipoRangoPalabras] of TipoFrec;

32         TipoCjtoFrecs = record
33         {
34              frecs: TipoFrecs;
35              numfrecs: TipoNatural;
36         };

38    function esblanco(c: char): bool
39    {
40         return c < 'a' or c > 'z';
41    }
```

```
43  procedure saltarblancos()
44        c: char;
45        fin: bool;
46  {
47        fin = False;
48        while(not fin){
49              peek(c);
50              switch(c){
51              case Eol:
52                    readeol();
53              case Eof:
54                    fin = True;
55              default:
56                    if(esblanco(c)){
57                          read(c);
58                    }else{
59                          fin = True;
60                    }
61              }
62        }
63  }

65  procedure leerpalabra(ref palabra: TipoPalabra)
66        c: char;
67  {
68        palabra.long = 0;
69        do{
70              read(c);
71              palabra.letras[palabra.long] = c;
72              palabra.long = palabra.long + 1;
73              peek(c);
74        }while(c != Eol and c != Eof and not esblanco(c));
75  }

77  procedure escribirpalabra(palabra: TipoPalabra)
78        i: TipoNatural;
79  {
80        for(i = 0, i < palabra.long){
81              write(palabra.letras[i]);
82        }
83  }

85  function longpalabra(palabra: TipoPalabra): TipoNatural
86  {
87        return palabra.long;
88  }
```

```
function igualpalabra(p1: TipoPalabra, p2: TipoPalabra): bool
     igual: bool;
     pos: TipoNatural;
{
     if(longpalabra(p1) != longpalabra(p2)){
          igual = False;
     }else{
          igual = True;
          pos = 0;
          while(pos < longpalabra(p1) and igual){
               igual = p1.letras[pos] == p2.letras[pos];
               pos = pos + 1;
          }
     }
     return igual;
}

function nuevapalabra(): TipoPalabra
     palabra: TipoPalabra;
{
     palabra.long = 0;
     return palabra;
}

procedure incrfrec(ref frec: TipoFrec)
{
     frec.veces = frec.veces + 1;
}

procedure escribirfrec(frec: TipoFrec)
{
     write("veces: ");
     writeln(frec.veces);
     write("palabra: ");
     escribirpalabra(frec.palabra);
     writeeol();
}

function nuevocjto(): TipoCjtoFrecs
     cjto: TipoCjtoFrecs;
{
     cjto.numfrecs = 0;
     return cjto;
}

function nuevafrec(palabra: TipoPalabra): TipoFrec
     frec: TipoFrec;
{
     frec.palabra = palabra;
     frec.veces = 1;
     return frec;
}
```

```
143  procedure buscar(c: TipoCjtoFrecs,
144                   palabra: TipoPalabra,
145                   ref encontrado: bool,
146                   ref pos: TipoNatural)
147  {
148       encontrado = False;
149       pos = 0;
150       while(pos < c.numfrecs and not encontrado){
151            if(igualpalabra(palabra, c.frecs[pos].palabra)){
152                 encontrado = True;
153            }else{
154                 pos = pos + 1;
155            }
156       }
157  }

159  procedure insertar(ref c: TipoCjtoFrecs, palabra: TipoPalabra)
160       pos: TipoNatural;
161       encontrado: bool;
162  {
163       buscar(c, palabra, encontrado, pos);
164       if(encontrado){
165            incrfrec(c.frecs[pos]);
166       }else{
167            c.numfrecs = c.numfrecs + 1;
168            c.frecs[pos] = nuevafrec(palabra);
169       }
170  }

172  procedure masfrecuente(c: TipoCjtoFrecs, ref palabra: TipoPalabra)
173       i: TipoNatural;
174       maxveces: TipoNatural;
175       maxpos: TipoNatural;
176  {
177       maxveces = 0;
178       maxpos = 0;
179       for(i = 0, i < c.numfrecs){
180            if(c.frecs[i].veces > maxveces){
181                 maxveces = c.frecs[i].veces;
182                 maxpos = i;
183            }
184       }
185       if(maxveces != 0){
186            palabra = c.frecs[maxpos].palabra;
187       }else{
188            palabra = nuevapalabra();
189       }
190  }

192  procedure escribircjto(cjto: TipoCjtoFrecs)
193       i: TipoNatural;
194  {
195       for(i = 0, i < cjto.numfrecs){
196            escribirfrec(cjto.frecs[i]);
197            writeeol();
198       }
199  }
```

```
201  procedure main()
202       palabra: TipoPalabra;
203       maslarga: TipoPalabra;
204       cjto: TipoCjtoFrecs;
205  {
206       cjto = nuevocjto();
207       do{
208            saltarblancos();
209            if(not eof()){
210                 leerpalabra(palabra);
211                 insertar(cjto, palabra);
212            }
213       }while(not eof());

215       masfrecuente(cjto, palabra);
216       escribirpalabra(palabra);
217       writeeol();
218  }
—
```

El programa incluye subprogramas para imprimir el conjunto. En general, debemos escribir los subprogramas para imprimir los tipos de datos de nuestros programas, para poder depurarlos y probarlos fácilmente.

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. La lectura se tiene que realizar carácter a carácter.

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. La lectura se tiene que realizar carácter a carácter.

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. Debe leerse el fichero carácter a carácter.

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. Debe leerse el fichero carácter a carácter. La entrada del programa ha de tomarse del fichero *datos.txt*.

19 Calcular derivadas de polinomios. Leer una serie de polinomios de la entrada y calcular 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. Debe leerse el fichero carácter a carácter. 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 cuándo 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 cuándo 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 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 arábiga 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

```
M0
```

para almacenar el último valor en la memoria 0 (vamos a numerar las memorias empezando por la posición 0) y

```
M1
```

para almacenarlo en la memoria 1. Para recuperar el valor de la memoria 0 podríamos utilizar `R0`, como en

```
3 / R0
```

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:

```
do{
       leerexpr(expr);
       if(not eof()){
              evalexpr(expr, valor);
              writeln(valor);
       }
}while(not eof());
```

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*:

```
types:
       TipoOp  = (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*:

```
consts:
       MaxNumArgs = 2;
types:
       TipoIndice = int 0..MaxNumArgs-1;
       TipoArgs = array[TipoIndice] of float;
```

Un programa nunca debe contener números ‘‘mágicos’’. Por ejemplo, nuestro *array* tiene dos elementos. Si vemos ese número por nuestro código, por ejemplo en una condición, podemos preguntarnos inmediatamente ¿Por qué 2 y no 5? Sin embargo, si vemos una constante, su nombre

nos dice qué es. 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.

|calc.p|

```
1     /*
2      *    Calculadora de linea de comandos
3      *    prototipo 1
4      */

6     program calc;



9     consts:
10          MaxNumArgs = 2;

12    types:
13          TipoOp  = (Suma, Resta, Multiplicacion, Division);
14          TipoIndice = int 0..MaxNumArgs-1;
15          TipoArgs = array[TipoIndice] of float;

17          /*
18           * expresion aritmetica de tipo "3 + 2"
19           */
20          TipoExpr = record
21          {
22               op: TipoOp;
23               args: TipoArgs;
24          };

26    procedure main()
27    {
28          writeln("nada");
29    }
```

—

¡Hora de compilar y ejecutar nuestro primer prototipo del programa!

```
; pick calc.p
; out.pam
nada
```

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, arg0: float, arg1: float): TipoExpr
2           expr: TipoExpr;
3     {
4           expr.op = op;
5           expr.args[0] = arg0;
6           expr.args[1] = arg1;
7           return expr;
8     }
```

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 de nuevo.

```
; pick calc.p
; out.pam
nada
```

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)
2     {
3          write(expr.args[0]);
4          escribirop(expr.op);
5          write(expr.args[1]);
6     }
```

Y ahora necesitamos...

```
1     procedure escribirop(op: TipoOp)
2     {
3          switch(op){
4          case Suma:
5               write('+');
6          case Resta:
7               write('-');
8          case Multiplicacion:
9               write('*');
10         case Division:
11              write('/');
12         }
13    }
```

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 programa principal:

```
1     procedure main()
2          prueba1: TipoExpr;
3          prueba2: TipoExpr;
4     {
5          prueba1 = nuevaexpr(Suma, 3.0, 2.2);
6          prueba2 = nuevaexpr(Division, 3.2, 0.0);
7          escribirexpr(prueba1);
8          writeeol();
9          escribirexpr(prueba2);
10         writeeol();
11    }
```

Y ahora lo probamos:

```
; pick calc.p
; out.pam
 3.000000+2.2000000
 3.200000+0.0000000
```

La suma y la resta salen junto a los operandos. Necesitamos algún espacio en blanco antes y después de escribir el operador. Para ello, hay que modificar *escribirexpr*. No cambiamos

escribirop para incluir espacios en blanco. Piensa que *escribirop* debe escribir una operación y nada más. Por tanto:

```
1     procedure escribirexpr(expr: TipoExpr)
2     {
3          write(expr.args[0]);
4          write(' ');
5          escribirop(expr.op);
6          write(' ');
7          write(expr.args[1]);
8     }
```

Ahora lo probamos de nuevo:

```
; pick calc.p
; out.pam
 3.000000 + 2.2000000
 3.200000 + 0.0000000
```

Mucho mejor. Dado que ya hemos probado estos subprogramas (aunque no mucho), es hora de borrar las pruebas y volver a dejar en el programa principal como sigue:

```
1     procedure main()
2     {
3          writeln("nada");
4     }
```

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): float
2          result: float;
3     {
4          result = 0.0;

6          switch(expr.op){
7          case Suma:
8               result = expr.args[0] + expr.args[1];
9          case Resta:
10              result = expr.args[0] - expr.args[1];
11         case Multiplicacion:
12              result = expr.args[0] * expr.args[1];
13         case Division:
14              result = expr.args[0] / expr.args[1];
15         }
16         return result;
17    }
```

Hora de probarlo. Escribimos de nuevo alguna prueba...

```
1          procedure main()
2               prueba1: TipoExpr;
3               prueba2: TipoExpr;
4               valor1: float;
5               valor2: float;
6          {
7               prueba1 = nuevaexpr(Suma, 3.0, 2.2);
8               prueba2 = nuevaexpr(Division, 3.2, 0.0);
9               valor1 = evalexpr(prueba1);
10              writeln(valor1);
11              valor2 = evalexpr(prueba2);
12              writeln(valor2);
13         }
```

Y la ejecutamos:

```
; pick calc.p
; out.pam
5.200000
calc.p:172: divide by zero
pc=0x3b7, sp=0xc8, fp=0xc8
```

Al intentar evaluar la segunda expresión de prueba ha ocurrido un error de ejecución: hemos dividido por cero. Aunque aquí no puede verse, la línea 172 del fichero *calc.p* cuando hicimos esta prueba era la línea de la función *evalexpr* que hemos mostrado antes como:

```
13              result = expr.args[0] / expr.args[1];
```

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

```
13              if(expr.args[1] == 0.0){
14                   ????
15              }else{
16                   arg1 = expr.args[1];
17              }
```

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     procedure evalexpr(expr: TipoExpr, ref result: float)
2          arg1: float;
3          nummem: int;
4     {
5          result = 0.0;
```

```
7          switch(expr.op){
8          case Suma:
9               result = expr.args[0] + expr.args[1];
10         case Resta:
11              result = expr.args[0] - expr.args[1];
12         case Multiplicacion:
13              result = expr.args[0] * expr.args[1];
14         case Division:
15              if(expr.args[1] == 0.0){
16                   arg1 = 1.0e-30;
17              }else{
18                   arg1 = expr.args[1];
19              }
20              result = expr.args[0] / arg1;
21         }
22    }
```

Si la probamos ahora parece que hace algo razonable (para este tipo de calculadora):

```
; pick calc.p
; out.pam
 5.20000
 320000004815189170000000000000.000000
```

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 *read* 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          procedure main()
2               expr: TipoExpr;
3               valor: float;
4          {
5               do{
6                    leerexpr(expr);
7                    if(not eof()){
8                         valor = evalexpr(expr);
9                         writeln(valor);
10                   }
11              }while(not eof());
12         }
```

El procedimiento que va a leer de la entrada es *leerexpr*. 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 un operando, saltamos blancos y leemos una operación, saltamos blancos y leemos otro operando. Claro, hay que tener cuidado de no seguir leyendo una vez hemos encontrado *Eof*. 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(ref expr: TipoExpr)
2     {
3          saltarblancos();
4          if(not eof()){
5               leernumero(expr.args[0]);
6          }
7          if(not eof()){
8               saltarblancos();
9          }
10         if(not eof()){
11              leerop(expr.op);
12         }
13         if(not eof()){
14              saltarblancos();
15         }
16         if(not eof()){
17              leernumero(expr.args[1]);
18         }
19    }
```

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.

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()* devuelve *True* múltiples veces. Compensa pagar el precio. De no haberlo hecho así, *leerexpr* habría quedado como sigue:

```
1     procedure leerexpr(ref expr: TipoExpr)
2     {
3           saltarblancos();
4           if(not eof()){
5                 leernumero(expr.args[0]);
6                 saltarblancos();
7                 if(not eof()){
8                       leerop(expr.op);
9                       saltarblancos();
10                      if(not eof()){
11                            leernumero(expr.args[1]);
12                      }
13                }
14
15          }
16    }
```

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(ref num: float)
2           c: char;
3           digito: int;
4     {
5           num = 0.0;

7           do{
8                 peek(c);
9                 if(esdigito(c)){
10                      read(c);
11                      digito = int(c) - int('0');
12                      num = num * 10.0 + float(digito);
13                }
14          }while(esdigito(c));
15    }
```

Dejamos de leer justo cuando dejamos de tener un dígito. 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(ref num: float)
2          c: char;
3          digito: int;
5          parteentera: bool;
6          fraccion: float;
7     {
8          num = 0.0;
9          parteentera = True;
10         fraccion = 1.0;

12         do{
13              peek(c);
14              if(esdigito(c)){
15                   read(c);
16                   digito = int(c) - int('0');
17                   if(parteentera){
18                        num = num * 10.0 + float(digito);
19                   }else{
20                        fraccion = fraccion / 10.0;
21                        num = num + float(digito) * fraccion;
22                   }
23              }
24         }while(esdigito(c));
25    }
```

Pero nos falta tener en cuenta cuándo pasamos de la parte entera a la parte decimal y 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(ref num: float)
2          c: char;
3          digito: int;
4          fin: bool;
5          parteentera: bool;
6          fraccion: float;
7          signo: float;
8     {
9          num = 0.0;
10         parteentera = True;
11         fraccion = 1.0;
12         signo = 1.0;
13         fin = False;

14         do{
15              peek(c);
16              switch(c){
17              case '-':
18                   signo = -1.0;
19                   read(c);
20              case '+':
21                   read(c);
22              case '.':
23                   parteentera = False;
24                   read(c);
```

```
25              case '0'..'9':
26                   read(c);
27                   digito = int(c) - int('0');
28                   if(parteentera){
29                        num = num * 10.0 + float(digito);
30                   }else{
31                        fraccion = fraccion / 10.0;
32                        num = num + float(digito) * fraccion;
33                   }

35              default:
36                   fin = True;
37              }

38         }while(not fin);
39         num = signo * num;
40    }
```

Fíjate que, tal y como está programado, este procedimiento acepta cosas raras, como que se repita varias veces el signo. Vamos a ignorar este hecho. Al menos el procedimiento lee números correctos de forma correcta y evita que el programa sufra errores cuando el usuario se equivoca al escribirlos.

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          TipoOp  = (Error, Suma, Resta, Multiplicacion, Division);
```

Y nuestro procedimiento para leer operaciones sería ahora:

```
1     procedure leerop(ref op: TipoOp)
2          c: char;
3     {
4          read(c);
5          switch(c){
6          case '+':
7               op = Suma;
8          case '-':
9               op = Resta;
10         case '*':
11              op = Multiplicacion;
12         case '/':
13              op = Division;
14         default:
15              op = Error;
16         }
17    }
```

¡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.

```
; out.pam
3 + 2
 5.0000000
-2
* 2
-4.000000
3 x 2
 0.000000
abc
 0.000000
3.5 / 0
 0.000000
3 / 1
 3.0000000
```

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.p

```
1    /*
2     *   Calculadora de linea de comandos
3     *   prototipo 2
4     */

6    program calc;

8    consts:
9         MaxNumArgs = 2;

11   types:
12        TipoOp  = (Error, Suma, Resta, Multiplicacion, Division);
13        TipoIndice = int 0..MaxNumArgs-1;
14        TipoArgs = array[TipoIndice] of float;

16        /*
17         *   expresion aritmetica de tipo "3 + 2"
18         */
19        TipoExpr = record
20        {
21             op: TipoOp;
22             args: TipoArgs;
23        };

25   function esblanco(c: char): bool
26   {
27        return c == ' ' or c == Tab;
28   }
```

```
procedure saltarblancos()
     c: char;
     fin: bool;
{
     fin = False;
     while(not fin){
          peek(c);
          switch(c){
          case Eol:
               readeol();
          case Eof:
               fin = True;
          default:
               if(esblanco(c)){
                    read(c);
               }else{
                    fin = True;
               }
          }
     }
}
procedure leerop(ref op: TipoOp)
     c: char;
{
     read(c);
     switch(c){
     case '+':
          op = Suma;
     case '-':
          op = Resta;
     case '*':
          op = Multiplicacion;
     case '/':
          op = Division;
     default:
          op = Error;
     }
}

procedure leernumero(ref num: float)
     c: char;
     digito: int;
     fin: bool;
     parteentera: bool;
     fraccion: float;
     signo: float;
{
     num = 0.0;
     parteentera = True;
     fraccion = 1.0;
     signo = 1.0;
     fin = False;
```

```
83          do{
84               peek(c);
85               switch(c){
86               case '-':
87                    signo = -1.0;
88                    read(c);
89               case '+':
90                    read(c);
91               case '.':
92                    parteentera = False;
93                    read(c);
94               case '0'..'9':
95                    read(c);
96                    digito = int(c) - int('0');
97                    if(parteentera){
98                         num = num * 10.0 + float(digito);
99                    }else{
100                        fraccion = fraccion / 10.0;
101                        num = num + float(digito) * fraccion;
102                   }
103              default:
104                   fin = True;
105              }
106         }while(not fin);
107         num = signo * num;
108    }

110    procedure leerexpr(ref expr: TipoExpr)
111    {
112         saltarblancos();
113         if(not eof()){
114              leernumero(expr.args[0]);
115         }
116         if(not eof()){
117              saltarblancos();
118         }
119         if(not eof()){
120              leerop(expr.op);
121         }
122         if(not eof()){
123              saltarblancos();
124         }
125         if(not eof()){
126              leernumero(expr.args[1]);
127         }
128    }

130    function nuevaexpr(op: TipoOp, arg0: float, arg1: float): TipoExpr
131         expr: TipoExpr;
132    {
133         expr.op = op;
134         expr.args[0] = arg0;
135         expr.args[1] = arg1;
136         return expr;
137    }
```

```
procedure escribirop(op: TipoOp)
{
     switch(op){
     case Suma:
          write('+');
     case Resta:
          write('-');
     case Multiplicacion:
          write('*');
     case Division:
          write('/');
     case Error:
          write("???");
     }
}

procedure escribirexpr(expr: TipoExpr)
{
     write(expr.args[0]);
     write(' ');
     escribirop(expr.op);
     write(' ');
     write(expr.args[1]);
}

function evalexpr(expr: TipoExpr): float
     result: float;
     arg1: float;
{
     result = 0.0;
     switch(expr.op){
     case Suma:
          result = expr.args[0] + expr.args[1];
     case Resta:
          result = expr.args[0] - expr.args[1];
     case Multiplicacion:
          result = expr.args[0] * expr.args[1];
     case Division:
          if(expr.args[1] == 0.0){
               arg1 = 1.0e-30;
          }else{
               arg1 = expr.args[1];
          }
          result = expr.args[0] / arg1;
     case Error:
          result = 0.0;
     }
     return result;
}
```

```
189   procedure main()
190         expr: TipoExpr;
191         valor: float;
192   {
193         do{
194               leerexpr(expr);
195               if(not eof()){
196                     valor = evalexpr(expr);
197                     writeln(valor);
198               }
199         }while(not eof());
200   }
```

—

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 otra constante...

```
1  consts:
2     MaxLongRom = 50;
```

para otro tipo de datos...

```
1  types:
2     TipoDigitoRom = (RomI, RomV, RomX, RomL, RomC, RomD, RomM);
3     TipoDigitosRom = array[0..MaxLongRom-1] of TipoDigitoRom;

5     TipoNumRom = record
6     {
7           digitos: TipoDigitosRom;
8           numdigitos: int;
9     };
```

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.

El plan es el siguiente. 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(ref expr: TipoExpr)
2     {
3           saltarblancos();
4           if(not eof()){
5                 leeroperando(expr.args[0]);
6           }
7           if(not eof()){
8                 saltarblancos();
9           }
10          if(not eof()){
11                leerop(expr.op);
12          }
13          if(not eof()){
14                saltarblancos();
15          }
16          if(not eof()){
17                leeroperando(expr.args[1]);
18          }
19    }
```

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(ref num: float)
2     {
3           if(hayromano()){
4                 leernumrom(num);
5           }else{
6                 leernumero(num);
7           }
8     }
```

No obstante, tenemos un tipo de datos que se ocupa de manipular números romanos: la llamada a *leernumrom* tiene mal aspecto. Podría quedar mucho mejor algo como leer un número romano y luego calcular su valor:

```
1     procedure leeroperando(ref num: float)
2          rom: TipoNumRom;
3     {
4          if(hayromano()){
5               leernumrom(rom);
6               num = valornumrom(rom);
7          }else{
8               leernumero(num);
9          }
10    }
```

¿Por qué? Sencillamente por que la llamada que teníamos antes se llamaba *leernumrom* pero leía 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 arábigos, lo cual es sorprendente. Podemos mirar el siguiente carácter de la entrada y decidir. Como una función no puede tener efectos laterales (y mirar en la entrada lo es), *hayromano* no puede ser una función. Por eso optamos por volver a modificar *leeroperando* y dejarlo como sigue:

```
1     procedure leeroperando(ref num: float, mem: TipoMem)
2          rom: TipoNumRom;
3          c: char;
4     {
5          peek(c);
6          if(esromano(c)){
7               leernumrom(rom);
8               num = valornumrom(rom);
9          }else{
10              leernumero(num);
11         }
12    }
```

Sabemos que *peek* no puede encontrar un fin de línea ni un fin de fichero (acabamos de saltar blancos) por lo que podemos ignorar estos casos.

Ver si un carácter tiene aspecto de pertenecer a un número romano es fácil:

```
1     function esromano(c: char): bool
2     {
3          switch(c){
4          case 'I', 'V', 'X', 'L', 'C', 'D', 'M':
5               return True;
6          default:
7               return False;
8          }
9     }
```

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(ref rom: TipoNumRom)
2          c: char;
3     {
4          rom.numdigitos = 0;  /* vacio por ahora */
5          do{
6               read(c);
7               rom.digitos[rom.numdigitos] = digitorom(c);
8               rom.numdigitos = rom.numdigitos + 1;
9               peek(c);
10         }while(esromano(c));
11    }
```

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: char): TipoDigitoRom
2          d: TipoDigitoRom;
3     {
4          switch(c){
5          case 'I':
6               d = RomI;
7          case 'V':
8               d = RomV;
9          case 'X':
10              d = RomX;
11         case 'L':
12              d = RomL;
13         case 'C':
14              d = RomC;
15         case 'D':
16              d = RomD;
17         default:
18              d = RomM;
19         }
20         return d;
21    }
```

En teoría, *c* puede ser cualquier carácter. En la práctica hemos comprobado si tenemos un dígito romano antes de leerlo. Para que el *switch* cubra todos los casos hemos escrito la última rama del *switch* utilizando *default* 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 el valor de un dígito es menor que el valor del que le sigue, entonces se resta su valor del total. 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): float
2          total: float;
3          valor: float;
4          sigvalor: float;
5          i: int;
6     {
7          total = 0.0;
8          for(i = 0, i < rom.numdigitos){
9               valor = ValorDigitoRom[rom.digitos[i]];
10              if(i < rom.numdigitos - 1){
11                   sigvalor = ValorDigitoRom[rom.digitos[i + 1]];
12                   if(valor < sigvalor){
13                        valor = - valor;
14                   }
15              }
16              total = total + valor;
17         }
18         return total;
19    }
```

Hemos tenido que tener cuidado con el último dígito. Además, como habrás podido ver si has leído el código, 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, de un nuevo tipo *TipoValorRom*:

```
1     types:
2          TipoValorRom = array[TipoDigitoRom] of float;
```

Y esta es la declaración del array constante:

```
1     consts:
2          ValorDigitoRom = TipoValorRom(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!

```
; out.pam
XIII + 3
 16.000000
```

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     TipoOp  = (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     /*
2      *   expresion aritmetica de tipo "3 + 2" o "sen 43"
3      */
4     TipoExpr = record
5     {
6          op: TipoOp;
7          args: TipoArgs;
8          numargs: int;
9     };
```

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, arg0: float, arg1: float): TipoExpr
2          expr: TipoExpr;
3     {
4          expr.op = op;
5          expr.numargs = 2;
6          expr.args[0] = arg0;
7          expr.args[1] = arg1;
8          return expr;
9     }

11    function nuevaexpr1(op: TipoOp, arg0: float): TipoExpr
12         expr: TipoExpr;
13    {
14         expr.op = op;
15         expr.numargs = 1;
16         expr.args[0] = arg0;
17         return expr;
18    }
```

Aunque la función *nuevaexpr* no se usaba, sí es importante hacer esto, puesto que ahora no es obvio cómo se crea una expresión. Es mejor arreglarlo para que si, en el futuro, queremos construir nuevas expresiones, podamos hacerlo sin sorpresas.

Lo mismo hay que hacer con *escribirop*, por cierto. Hay otras operaciones nuevas. 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)
2     {
3          if(expr.numargs == 1){
4               escribirop(expr.op);
5               write(' ');
6               write(expr.args[0]);
7          }else{
8               write(expr.args[0]);
9               write(' ');
10              escribirop(expr.op);
11              write(' ');
12              write(expr.args[1]);
13         }
14    }
```

La función *evalexpr* debe ser capaz de evaluar las nuevas expresiones. Basta añadir más casos al *case*:

```
19         case Seno:
20              result = sin(expr.args[0]);
21         case Coseno:
22              result = cos(expr.args[0]);
23         case Tangente:
24              result = tan(expr.args[0]);
25         case Raiz:
26              result = sqrt(expr.args[0]);
```

Las funciones elementales, como *sin*, son funciones predefinidas como parte del lenguaje.

Ahora tenemos que cambiar la forma en que leemos expresiones. La idea es que nuestro actual *leerexpr* es en realidad un procedimiento que lee una expresión binaria, no una expresión en general. Podemos renombrarlo para que quede claro que ahora se dedica sólo a leer expresiones binarias. Nos faltaría otro procedimiento para leer una expresión unaria y un nuevo *leerexpr* que vea cuál de los dos tiene que usar. Lo que determina si tenemos que usar uno u otro es ver si, al principio de una expresión en la entrada, tenemos una función o tenemos un número (que puede ser romano o arábigo).

Cambiamos pues *leerexpr* para que sea:

```
1     procedure leerexpr(ref expr: TipoExpr)
2          c: char;
3          fin: bool;
4     {
5          saltarblancos();
6          if(not eof()){
7               peek(c);
8               if(esfuncion(c)){
9                    leerexpr1(expr);
10              }else{
11                   leerexpr2(expr);
12              }
13         }
14    }
```

Podemos implementar *esfuncion* y olvidarnos de ella.

```
1     function esfuncion(c: char): bool
2     {
3          return c >= 'a' and c <= 'z';
4     }
```

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. ¡Por eso los llamamos *leerexpr1* y *leerexpr2*!, por el número de argumentos de la expresión que leen. Para hacer *leerexpr1*, lo programamos de un modo similar a *leerexpr2*:

```
1          procedure leerexpr1(ref expr: TipoExpr)
2          {
3                leerop(expr.op);
4                saltarblancos();
5                if(not eof()){
6                     leeroperando(expr.args[0]);
7                     expr.numargs = 1;
8                }
9          }
```

Hay un cambio que tenemos que hacer en *leerexpr2*: debe inicializar *expr.numargs* a 2, para indicar que se ha leído una expresión binaria. Recuerda que cuando lo programamos todas las expresiones eran binarias, pero ahora no.

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 él. Además, volvemos a tomar prestado el procedimiento *leerpalabra* y la función *igualpalabra* (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(ref op: TipoOp)
2          palabra: TipoPalabra;
3     {
4          leerpalabra(palabra);
5          if(igualpalabra(palabra, NombreSuma)){
6               op = Suma;
7          }else if(igualpalabra(palabra, NombreResta)){
8               op = Resta;
9          }else if(igualpalabra(palabra, NombreMultiplicacion)){
10              op = Multiplicacion;
11         }else if(igualpalabra(palabra, NombreDivision)){
12              op = Division;
13         }else if(igualpalabra(palabra, NombreSeno)){
14              op = Seno;
15         }else if(igualpalabra(palabra, NombreCoseno)){
16              op = Coseno;
17         }else if(igualpalabra(palabra, NombreTangente)){
18              op = Tangente;
19         }else if(igualpalabra(palabra, NombreRaiz)){
20              op = Raiz;
21         }else{
22              op = Error;
23         }
24    }
```

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 cambiarlo si así lo queremos. Es conveniente recordar que las constantes son de tipo *TipoPalabra*, que contiene un array de tipo *TipoLetras*. Este array es de longitud *TipoRangoPalabra* y hay que inicializar todos sus valores. Hemos rellenado los huecos restantes con puntos (el programa luego ignora estos valores, dado que el segundo campo de *TipoPalabra* indica cuántos caracteres hay que considerar).

```
1         NombreSuma           = TipoPalabra("+.........", 1);
2         NombreResta          = TipoPalabra("-.........", 1);
3         NombreMultiplicacion = TipoPalabra("*.........", 1);
4         NombreDivision       = TipoPalabra("/.........", 1);
5         NombreSeno           = TipoPalabra("sen.......", 3);
6         NombreCoseno         = TipoPalabra("cos.......", 3);
7         NombreTangente       = TipoPalabra("tan.......", 3);
8         NombreRaiz           = TipoPalabra("raiz......", 4);
```

Ahora podemos utilizar nuestra calculadora como se ve a continuación:

```
; out.pam
sen MCM
 0.615922
3 - XI
-8.000000
raiz X
 3.162278
```

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 cabecera 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 ‘‘`M0`’’ y ‘‘`M1`’’ 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

```
mem0
```

y

```
mem1
```

Pero, como podrá verse, es igual de fácil admitir operaciones de la forma

```
mem 0
```

o

```
mem 1
```

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 ‘‘R0’’ o ‘‘R1’’ 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     TipoOp  = (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     }else if(igualpalabra(palabra, NombreGuardar)){
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 = TipoPalabra("mem.......", 3);
```

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 que podemos encontrar en la entrada: ‘‘mem’’.

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 una nueva constante:

```
1     consts:
2           MaxNumMem = 2;
```

que usaremos para un nuevo tipo:

```
1     types:
2           TipoRangoMem = int 0..MaxNumMem-1;
3           TipoMem = array[TipoRangoMem] of float;
```

Y ahora ajustamos un poco el programa principal:

```
1     procedure main()
2           expr: TipoExpr;
3           valor: float;
4           mem: TipoMem;

5     {
6           valor = 0.0;
7           nuevamem(mem);
```

```
8          do{
9               leerexpr(expr, mem);
10              if(not eof()){
11                   evalexpr(expr, valor, mem);
12                   writeln(valor);
13              }
14         }while(not eof());
15    }
```

Ahora empezamos inicializando a cero tanto el último valor calculado como la memoria. . El procedimiento *nuevamem* inicializa las memorias a 0. 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, ref result: float, ref mem: TipoMem)
```

En caso de que la operación sea guardar, procedemos de este modo en *evalexpr*:

```
1          case Guardar:
2               nummem = int(expr.args[0]);
3               if(nummem >= 0 and nummem < MaxNumMem){
4                    mem[nummem] = result;
5               }
```

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 ‘‘`R0`’’ fuese el valor de la memoria 0 y ‘‘`R1`’’ el valor de la memoria 1. 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 *leeroperando* 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(ref num: float, mem: TipoMem)
2          rom: TipoNumRom;
3          c: char;
4     {
5          peek(c);
6          if(esromano(c)){
7               leernumrom(rom);
8               num = valornumrom(rom);
9          }else if(esmemoria(c)){
10              leernummem(num, mem);
11         }else{
12              leernumero(num);
13         }
14    }
```

leernummem utiliza la memoria para dejar en *num* el contenido de la memoria cuyo nombre se encuentra en la entrada estándar.

```
1     procedure leernummem(ref num: float, mem: TipoMem)
2          c: char;
3          fin: bool;
4          digito: int;
5     {
6          fin = False;
7          read(c);   /* 'R' */
8          read(c);   /* cualquiera sabe lo que leemos aqui... */
9          num = 0.0;
10         if(esdigito(c)){
11              digito = int(c) - int('0');
12              if(digito >= 0 and digito < MaxNumMem){
13                   num = mem[digito];
14              }
15         }
16    }
```

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 *mem* como argumento 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.p|

```
1     /*
2      *   Calculadora de linea de comandos
3      *   prototipo 3
4      */

6     program calc;

8     consts:
9          MaxNumArgs = 2;
10         MaxLongRom = 50;
11         MaxLongPalabra = 10;
12         MaxNumMem = 2;

14    types:
15         TipoOp  = (Error, Suma, Resta, Multiplicacion,
16                    Division, Seno, Coseno, Tangente, Raiz, Guardar);
17         TipoIndice = int 0..MaxNumArgs-1;
18         TipoArgs = array[TipoIndice] of float;
```

```
20          /*
21           *    expresion aritmetica de tipo "3 + 2" o "sen 43"
22           */
23          TipoExpr = record
24          {
25               op: TipoOp;
26               args: TipoArgs;
27               numargs: int;
28          };

30          TipoDigitoRom = (RomI, RomV, RomX, RomL, RomC, RomD, RomM);
31          TipoValorRom = array[TipoDigitoRom] of float;

33          TipoDigitosRom = array[0..MaxLongRom-1] of TipoDigitoRom;
34          TipoNumRom = record
35          {
36               digitos: TipoDigitosRom;
37               numdigitos: int;
38          };

40          TipoRangoPalabra = int 0..MaxLongPalabra-1;
41          TipoLetras = array[TipoRangoPalabra] of char;
42          TipoPalabra = record
43          {
44               letras: TipoLetras;
45               long: int;
46          };

48          TipoRangoMem = int 0..MaxNumMem-1;
49          TipoMem = array[TipoRangoMem] of float;

51     consts:
52          ValorDigitoRom = TipoValorRom(1.0, 5.0, 10.0, 50.0, 100.0, 500.0, 1000.0);

54          NombreSuma           = TipoPalabra("+.........", 1);
55          NombreResta          = TipoPalabra("-.........", 1);
56          NombreMultiplicacion = TipoPalabra("*.........", 1);
57          NombreDivision       = TipoPalabra("/.........", 1);
58          NombreSeno           = TipoPalabra("sen.......", 3);
59          NombreCoseno         = TipoPalabra("cos.......", 3);
60          NombreTangente       = TipoPalabra("tan.......", 3);
61          NombreRaiz           = TipoPalabra("raiz......", 4);
62          NombreGuardar        = TipoPalabra("mem.......", 3);


65     function esblanco(c: char): bool
66     {
67          return c == ' ' or c == Tab;
68     }
```

```
70   function esromano(c: char): bool
71   {
72        switch(c){
73        case 'I', 'V', 'X', 'L', 'C', 'D', 'M':
74             return True;
75        default:
76             return False;
77        }
78   }

80   function esfuncion(c: char): bool
81   {
82        return c >= 'a' and c <= 'z';
83   }

85   function esmemoria(c: char): bool
86   {
87        return c == 'R';
88   }

90   procedure saltarblancos()
91        c: char;
92        fin: bool;
93   {
94        fin = False;
95        while(not fin){
96             peek(c);
97             switch(c){
98             case Eol:
99                  readeol();
100            case Eof:
101                 fin = True;
102            default:
103                 if(esblanco(c)){
104                      read(c);
105                 }else{
106                      fin = True;
107                 }
108            }
109       }
110  }
111  procedure leerpalabra(ref palabra: TipoPalabra)
112       c: char;
113  {
114       palabra.long = 0;    /* vacia por ahora*/
115       do{
116            read(c);
117            palabra.letras[palabra.long] = c;
118            palabra.long = palabra.long + 1;
119            peek(c);
120       }while(c != Eol and c != Eof and not esblanco(c));
121  }
```

```
function igualpalabra(p1: TipoPalabra, p2: TipoPalabra): bool
     iguales: bool;
     i: TipoRangoPalabra;
{
     if(p1.long == p2.long){
          iguales = True;
          i = 0;
          while(i < p1.long and iguales){
               iguales = p1.letras[i] == p2.letras[i];
               i = i + 1;
          }
     }else{
          iguales = False;
     }
     return iguales;
}


procedure leerop(ref op: TipoOp)
     palabra: TipoPalabra;
{
     leerpalabra(palabra);
     if(igualpalabra(palabra, NombreSuma)){
          op = Suma;
     }else if(igualpalabra(palabra, NombreResta)){
          op = Resta;
     }else if(igualpalabra(palabra, NombreMultiplicacion)){
          op = Multiplicacion;
     }else if(igualpalabra(palabra, NombreDivision)){
          op = Division;
     }else if(igualpalabra(palabra, NombreSeno)){
          op = Seno;
     }else if(igualpalabra(palabra, NombreCoseno)){
          op = Coseno;
     }else if(igualpalabra(palabra, NombreTangente)){
          op = Tangente;
     }else if(igualpalabra(palabra, NombreRaiz)){
          op = Raiz;
     }else if(igualpalabra(palabra, NombreGuardar)){
          op = Guardar;
     }else{
          op = Error;
     }
}

procedure leernumero(ref num: float)
     c: char;
     digito: int;
     fin: bool;
     parteentera: bool;
     fraccion: float;
     signo: float;
{
     num = 0.0;
     parteentera = True;
     fraccion = 1.0;
     signo = 1.0;
     fin = False;
```

```
183         do{
184               peek(c);
185               switch(c){
186               case '-':
187                     signo = -1.0;
188                     read(c);
189               case '+':
190                     read(c);

192               case '.':
193                     parteentera = False;
194                     read(c);

196               case '0'..'9':
197                     read(c);
198                     digito = int(c) - int('0');
199                     if(parteentera){
200                           num = num * 10.0 + float(digito);
201                     }else{
202                           fraccion = fraccion / 10.0;
203                           num = num + float(digito) * fraccion;
204                     }

206               default:
207                     fin = True;
208               }
209         }while(not fin);
210         num = signo * num;
211   }

213   function digitorom(c: char): TipoDigitoRom
214         d: TipoDigitoRom;
215   {
216         switch(c){
217         case 'I':
218               d = RomI;
219         case 'V':
220               d = RomV;
221         case 'X':
222               d = RomX;
223         case 'L':
224               d = RomL;
225         case 'C':
226               d = RomC;
227         case 'D':
228               d = RomD;
229         default:
230               d = RomM;
231         }
232         return d;
233   }
```

```
235  procedure leernumrom(ref rom: TipoNumRom)
236       c: char;
237  {
238       rom.numdigitos = 0;  /* vacio por ahora */
239       do{
240            read(c);
241            rom.digitos[rom.numdigitos] = digitorom(c);
242            rom.numdigitos = rom.numdigitos + 1;
243            peek(c);
244       }while(esromano(c));
245  }

247  function valornumrom(rom: TipoNumRom): float
248       total: float;
249       valor: float;
250       sigvalor: float;
251       i: int;
252  {
253       total = 0.0;
254       for(i = 0, i < rom.numdigitos){
255            valor = ValorDigitoRom[rom.digitos[i]];
256            if(i < rom.numdigitos - 1){
257                 sigvalor = ValorDigitoRom[rom.digitos[i + 1]];
258                 if(valor < sigvalor){
259                      valor = - valor;
260                 }
261            }
262            total = total + valor;
263       }
264       return total;
265  }

267  function esdigito(c: char): bool
268  {
269       return c >= '0' and c <= '9';
270  }

272  procedure leernummem(ref num: float, mem: TipoMem)
273       c: char;
274       fin: bool;
275       digito: int;
276  {
277       fin = False;
278       read(c);   /* 'R' */
279       read(c);   /* cualquiera sabe lo que leemos aqui... */
280       num = 0.0;
281       if(esdigito(c)){
282            digito = int(c) - int('0');
283            if(digito >= 0 and digito < MaxNumMem){
284                 num = mem[digito];
285            }
286       }
287  }
```

```
289  procedure leeroperando(ref num: float, mem: TipoMem)
290       rom: TipoNumRom;
291       c: char;
292  {
293       peek(c);
294       if(esromano(c)){
295            leernumrom(rom);
296            num = valornumrom(rom);
297       }else if(esmemoria(c)){
298            leernummem(num, mem);
299       }else{
300            leernumero(num);
301       }
302  }

304  procedure leerexpr2(ref expr: TipoExpr, mem: TipoMem)
305  {
306       leeroperando(expr.args[0], mem);
307       if(not eof()){
308            saltarblancos();
309       }
310       if(not eof()){
311            leerop(expr.op);
312       }
313       if(not eof()){
314            saltarblancos();
315       }
316       if(not eof()){
317            leeroperando(expr.args[1], mem);
318            expr.numargs  = 2;
319       }
320  }

322  procedure leerexpr1(ref expr: TipoExpr, mem: TipoMem)
323  {
324       leerop(expr.op);
325       if(not eof()){
326            saltarblancos();
327       }
328       if(not eof()){
329            leeroperando(expr.args[0], mem);
330            expr.numargs  = 1;
331       }
332  }
```

```
334  procedure leerexpr(ref expr: TipoExpr, mem: TipoMem)
335       c: char;
336       fin: bool;
337  {
338       saltarblancos();
339       if(not eof()){
340            peek(c);
341            if(esfuncion(c)){
342                 leerexpr1(expr, mem);
343            }else{
344                 leerexpr2(expr, mem);
345            }
346       }
347  }

349  procedure escribirpalabra(palabra: TipoPalabra)
350       i: TipoRangoPalabra;
351  {
352       for(i = 0, i < palabra.long){
353            write(palabra.letras[i]);
354       }
355  }

357  procedure escribirop(op: TipoOp)
358  {
359       switch(op){
360       case Suma:
361            escribirpalabra(NombreSuma);
362       case Resta:
363            escribirpalabra(NombreResta);
364       case Multiplicacion:
365            escribirpalabra(NombreMultiplicacion);
366       case Division:
367            escribirpalabra(NombreDivision);
368       case Seno:
369            escribirpalabra(NombreSeno);
370       case Coseno:
371            escribirpalabra(NombreCoseno);
372       case Tangente:
373            escribirpalabra(NombreTangente);
374       case Raiz:
375            escribirpalabra(NombreRaiz);
376       case Error:
377            write("???");
378       }
379  }

381  function nuevaexpr1(op: TipoOp, arg0: float): TipoExpr
382       expr: TipoExpr;
383  {
384       expr.op = op;
385       expr.numargs = 1;
386       expr.args[0] = arg0;
387       return expr;
388  }
```

```
function nuevaexpr2(op: TipoOp, arg0: float, arg1: float): TipoExpr
     expr: TipoExpr;
{
     expr.op = op;
     expr.numargs = 2;
     expr.args[0] = arg0;
     expr.args[1] = arg1;
     return expr;
}

procedure escribirexpr(expr: TipoExpr)
{
     if(expr.numargs == 1){
          escribirop(expr.op);
          write(' ');
          write(expr.args[0]);
     }else{
          write(expr.args[0]);
          write(' ');
          escribirop(expr.op);
          write(' ');
          write(expr.args[1]);
     }
}
```

```
415  procedure evalexpr(expr: TipoExpr, ref result: float, ref mem: TipoMem)
416       arg1: float;
417       nummem: int;
418  {
419       switch(expr.op){
420       case Suma:
421            result = expr.args[0] + expr.args[1];
422       case Resta:
423            result = expr.args[0] - expr.args[1];
424       case Multiplicacion:
425            result = expr.args[0] * expr.args[1];
426       case Division:
427            if(expr.args[1] == 0.0){
428                 arg1 = 1.0e-30;
429            }else{
430                 arg1 = expr.args[1];
431            }
432            result = expr.args[0] / arg1;
433       case Seno:
434            result = sin(expr.args[0]);
435       case Coseno:
436            result = cos(expr.args[0]);
437       case Tangente:
438            result = tan(expr.args[0]);
439       case Raiz:
440            result = sqrt(expr.args[0]);
441       case Guardar:
442            nummem = int(expr.args[0]);
443            if(nummem >= 0 and nummem < MaxNumMem){
444                 mem[nummem] = result;
445            }
446       case Error:
447            result = 0.0;
448       }
449  }

451  procedure nuevamem(ref mem: TipoMem)
452       i: int;
453  {
454       for(i = 0, i < MaxNumMem){
455            mem[i] = 0.0;
456       }
457  }
```

```
459   procedure main()
460         expr: TipoExpr;
461         valor: float;
462         mem: TipoMem;
463   {
464         valor = 0.0;
465         nuevamem(mem);
466         do{
467               leerexpr(expr, mem);
468               if(not eof()){
469                     evalexpr(expr, valor, mem);
470                     writeln(valor);
471               }
472         }while(not eof());
473   }
—
```

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

En general, las variables (y constantes) 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. En este curso no se permite el uso de variables globales. Es más, por omisión, Picky no te dejará compilar si tienes variables globales. Estas variables son peligrosas, ya que son accesibles desde cualquier subprograma, y no son recomendables para aprender a programar. Las constantes que hemos estado usando en todos nuestros programas son globales.

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 en tiempo de ejecución, 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 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’’; una variable estática se denomina así puesto que su gestión se puede decidir en tiempo de compilación). Este trozo de la memoria que pedimos en tiempo de ejecución 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. Podemos declarar una variable que sea una cadena de caracteres (un *array* de *char*) de cierto tamaño. Cuando el programa ejecute, ese *array* siempre tendrá ese tamaño. Da igual cómo ejecute el programa, la memoria usada por el *array* siempre es la misma. Con variables dinámicas, la idea es que durante la ejecución del programa podemos ver cuánta memoria necesitamos para almacenar nuestro *array*, y pedir justo esa nueva cantidad de memoria para un nuevo *array*. Cuando el *array* 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 *array* siempre tiene el mismo tamaño; en el segundo caso (dinámico) tendrá uno u otro en función de cuánta memoria pidamos en **tiempo de ejecución** (mientras que el programa ejecuta). Piensa también que en el primer caso el *array* 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 es sencillo, suele resultar confuso la primera vez que se ve, y requiere algo de esfuerzo 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 *int*, podemos declarar un tipo de datos que sea ‘‘puntero a *int*’’. Un puntero a *int* podrá apuntar a cualquier zona de la memoria en la que tengamos un *int*. La idea entonces es que, si en tiempo de ejecución queremos un *nuevo* entero, podemos pedir nueva memoria para un *int* y utilizar un puntero a *int* para referirnos a ella.

Con un ejemplo se verá todo mas claro. En Picky se declaran tipos de datos puntero usando el símbolo ‘‘^’’ antes del tipo apuntado. Se usa dicho símbolo porque recuerda a una flecha. Por ejemplo, esto declara un nuevo tipo de datos para representar punteros a enteros:

```
types:
       TipoPtrEntero = ^int;
```

Igualmente, esto declara un tipo de datos para punteros a datos de tipo *TipoPalabra*:

```
TipoPtrPalabra = ^TipoPalabra;
```

Las variables de tipo *TipoPtrEntero* pueden o bien **apuntar** a una variable de tipo *int* o bien tener el valor especial **nil**. Este es un valor nulo que representa que el puntero no apunta a ningún sitio. También se le suele llamar (en otros lenguajes) *null*. Dicho valor es compatible con cualquier tipo que sea un puntero.

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.

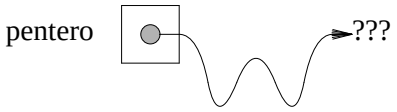


Figura 11.1: *Puntero recién declarado. No sabemos dónde apunta.*

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 *nil*, que hace que el puntero no apunte a ningún sitio.

```
pentero = nil;
```

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).

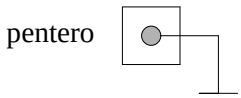


Figura 11.2: *Ejemplo de puntero a nil. No apunta a ningún sitio.*

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 Picky por nosotros. Pero ahora, teniendo un puntero a entero, podemos crear un nuevo entero y utilizar el puntero para referirnos a el. Esta sentencia

```
new(pentero);
```

crea una nueva variable de tipo *int* (sin nombre alguno) y hace que su dirección se le asigne a *pentero*. El efecto es el que muestra la figura 11.3.

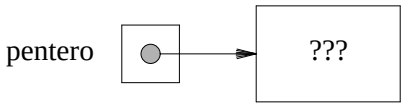


Figura 11.3: *Un puntero apuntando a un nuevo entero*

Ahora el puntero apunta a un nuevo entero. Eso sí, como no hemos inicializado el entero no sabemos qué valor tiene el entero. Puede tener cualquier valor.

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 Picky, para referirnos a la variable a la que apunta un puntero tenemos que añadir ‘‘^’’ tras el nombre del puntero. Esta sentencia inicializa el entero para que su valor sea 4:

```
pentero^ = 4;
```

El resultado puede verse en la figura 11.4.

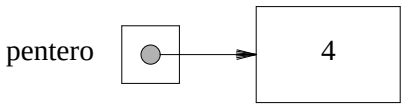


Figura 11.4: *Un puntero que apunta a nuestro entero ya inicializado.*

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^* cuando el valor de *pentero* es *nil* provoca la detención del programa con una indicación de error.

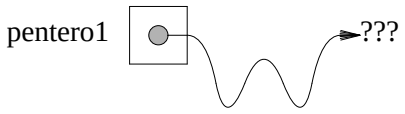
Utilizar un puntero cuando no está inicializado es aún peor: no sabemos dónde apunta el puntero y podemos estar accediendo (¡al azar!) a cualquier parte de la memoria. El resultado es muy parecido a un poltergueist. En muchas ocasiones, el compilador de Picky nos protegerá de este tipo de errores porque nos obliga a inicializar las variables. Pero no en todos los casos. Tienes que ser muy cuidadoso, estos errores son los más difíciles de arreglar.

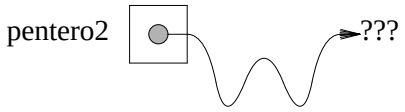
Respecto al estilo, los nombres de tipos puntero deberían ser siempre identificadores que comiencen por *TipoPtr* para indicar que se trata de un tipo de puntero. Igualmente, los nombres de variables de tipo puntero deberían dejar claro que son un puntero; por ejemplo, pueden ser siempre identificadores 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:

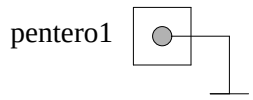
```
pentero1: TipoPtrEntero;
pentero2: TipoPtrEntero;
```

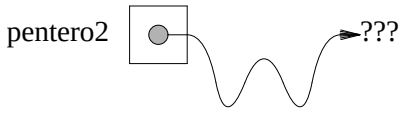




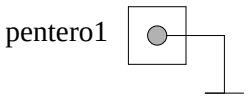
Y ahora ejecutemos algunas sentencias...

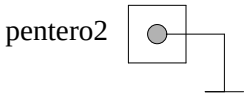
```
pentero1 = nil;
```



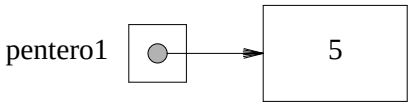


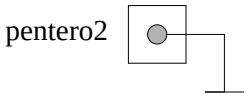
```
pentero2 = pentero1;
```



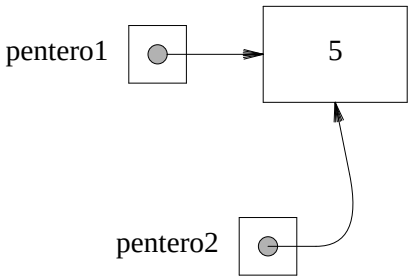



```
new(pentero1);
pentero1^ = 5;
```

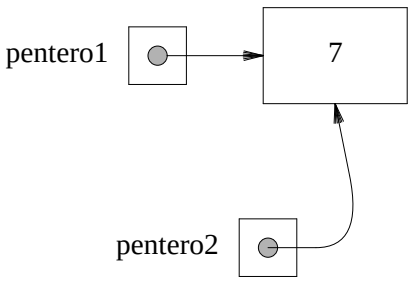





```
pentero2 = pentero1;
```




```
pentero2^ = 7;
```



Si en este punto ejecutamos

```
write(pentero1^);
```

veríamos que el valor de *pentero1^* es 7. Habrás visto que esta sentencia

```
pentero1^ = pentero2^;
```

copia el valor al que apunta *puntero2* a la variable a la que apunta *puntero1*, pero no modifica ningún puntero. En cambio

```
pentero1 = pentero2;
```

modifica *pentero1* y lo hace apuntar allí donde apunte *pentero2*.

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. Si nunca liberamos memoria, al final se nos acabará. El programa tiene que liberar todos los recursos que ha reservado cuando ya no los necesita.

De igual modo que ejecutamos

```
new(pentero);
```

cuando queremos crear un nuevo entero, tenemos que destruir dicho entero cuando ya no nos resulte útil. La forma de hacer esto en Picky es utilizar el procedimiento *dispose*. Este procedimiento, igual que el procedimiento *new*, admite como parámetro cualquier tipo que sea un puntero a otro tipo de datos.

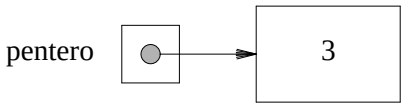
Por ejemplo, este programa crea una variable dinámica para guardar un entero en ella, entonces inicializa la variable a 3, imprime el contenido de la variable y, por último, libera la memoria dinámica solicitada anteriormente.

liberar.p

```
1   /*
2    *    Reserva y liberacion de memoria dinamica
3    */

5   program liberar;

7   types:
8        TipoPtrEntero= ^int;

10  procedure main()

12       pentero: TipoPtrEntero;
13  {
14       new(pentero);
15       pentero^ = 3;
16       writeln(pentero^);
```

```
18          /*
19           * liberamos la memoria de la variable a
20           * la que apunta el puntero, no la usamos mas
21           */
22          dispose(pentero);
23          pentero = nil;
24      }
```

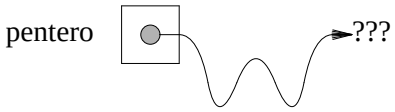
Antes de llamar a *dispose*, el estado de la memoria podría ser como se ve aquí:



La llamada

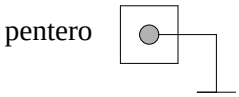
```
dispose(pentero);
```

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:

```
pentero = nil;
```



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.

Picky detecta este tipo de error y provoca un fallo de ejecución. En muchos otros lenguajes de programación esto no pasa. Si aplicamos una indirección, se accede a esa posición de memoria, haya lo que haya allí en ese momento. 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 (como Picky) 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 *nil* 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.

Si al acabar la ejecución del programa Picky detecta que se ha dejado memoria sin liberar, el programa acaba con un error de ejecución. Por ejemplo, si quitamos el *dispose* del programa anterior, al compilar y ejecutar veremos este error:

```
; pick liberar.p
; out.pam
3
memory leaks:
liberar.p:14 (1 times)
```

El error nos dice que tenemos un **leak** (en inglés, gotera) de memoria. ¡O que perdemos memoria! Esto es, que nos hemos dejado memoria dinámica sin liberar. El error también nos dice que la memoria que hemos pedido en la línea 14 de *liberar.p* la hemos perdido (no la hemos liberado) una vez. Ten en cuenta que muchos lenguajes *no* te van a ayudar a detectar cuándo dejas de liberar memoria. Debes hacerlo tu solo.

En este curso es necesario liberar toda la memoria antes de acabar.

Algunos lenguajes, entre los que **no** se encuentra Picky, inicializan casi todos los punteros (aquellos que no son parámetros) a *nil* y también se encargan de liberar la memoria dinámica cuando no quedan punteros apuntando hacia ella, sin que tengamos que llamar a *dispose* o ninguna otra función. A esta última habilidad se la llama **recolección de basura**. Muchos lenguajes notables, por ejemplo Pascal, C y C++, no tienen esta habilidad. Otros la tienen sólo a medias, por lo que no puedes confiar en que el lenguaje de programación solucione estos problemas.

11.6. Punteros a registros y arrays

Podemos tener punteros a cualquier tipo de datos, incluidos los registros. Por ejemplo, dadas la siguientes declaraciones:

```
1     types:
2          TipoNodo = record
3          {
4               valor: int;
5               peso: int;
6          };
7          TipoPtrNodo = ^TipoNodo;

9          TipoCadena = array[0..9] of char;
10         TipoPtrCadena = ^TipoCadena;
```

Después podremos declarar dos variables:

```
pnodo: TipoPtrNodo;
pcadena: TipoPtrCadena;
```

Y suponiendo que hemos ejecutado:

```
new(pnodo);
new(pcadena);
```

para asignar un 3 al campo *valor* del nodo al que apunta *pnodo* podemos escribir:

```
pnodo^.valor = 3;
```

Para inicializar la cadena a la que apunta *pcadena* con una ‘‘x’’ en todos sus caracteres, podemos hacer esto:

```
for(i = 0, i < len TipoCadena){
       pcadena^[i] = 'x';
}
```

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.

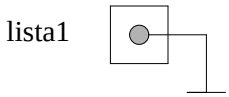
Intenta prestar atención a cómo se utilizan los punteros. Cómo sea una lista es algo de lo que aún no deberías preocuparte (aunque te hará falta en el futuro) . Una vez domines la técnica de la programación podrás aprender más sobre algoritmos y sobre estructuras de datos. Pero hay que aprender a andar antes de empezar a correr.

Una lista se define como algo que es una de estas dos cosas:

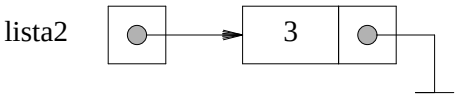
1 Una lista vacía. Esto es, nada.

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.

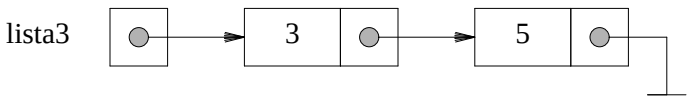
Supongamos que tenemos una lista de enteros. Esto sería una lista vacía:



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 Picky para una lista podría ser como sigue:

```
7     types:
8           TipoListaEnteros = ^TipoNodoEntero;

10          TipoNodoEntero = record
11          {
12                valor: int;
13                siguiente: TipoListaEnteros;
14          };
```

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 de la lista, *TipoListaEnteros*.

Aquí cada tipo de datos (lista y nodo) hace referencia al otro: es una definición circular. Picky nos permite hacer referencia a un tipo todavía no definido para romper la circularidad, pero sólo para tipos que son punteros. Posteriormente hay que declarar dicho tipo, claro está.

Crear una lista vacía es tan simple como inicializar a *nil* una variable de tipo *TipoListaEnteros.* Esta función hace el trabajo.

```
1    function listaenterosvacia(): TipoListaEnteros
2         l: TipoListaEnteros;
3    {
4         l = nil;
5         return l;
6    }
```

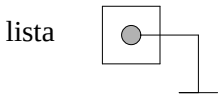
Si tenemos una variable

```
lista: TipoListaEnteros;
```

y la inicializamos así

```
lista = listaenterosvacia();
```

entonces esta variable quedará...



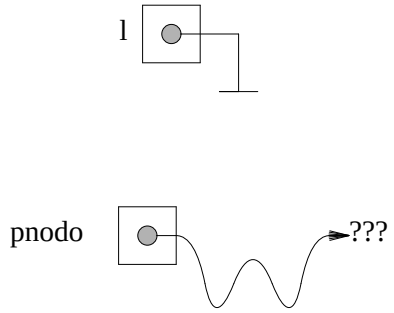
Para ver si una lista es vacía podemos implementar una función como esta:

```
1    function eslistaenterosvacia(l: TipoListaEnteros): bool
2    {
3         return l == nil;
4    }
```

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.

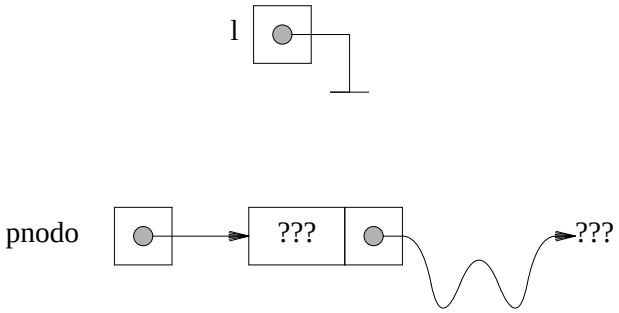
```
1    procedure insertarentero(ref l: TipoListaEnteros, n: int)
2         pnodo: TipoListaEnteros;
3    {
4         new(pnodo);
5         pnodo^.valor = n;
6         pnodo^.siguiente = l;
7         l = pnodo;
8    }
```

Si llamamos a este procedimiento para insertar un 3 en una lista vacía, tendríamos este estado justo cuando estamos en ‘‘{’’, al principio del procedimiento:



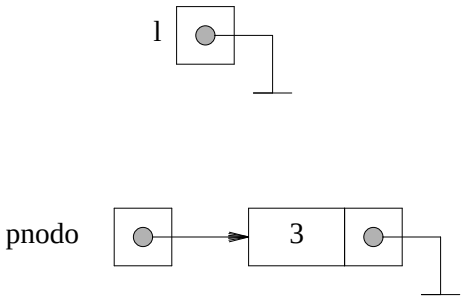
Tras la primera sentencia del procedimiento tenemos:

```
4       new(pnodo);
```



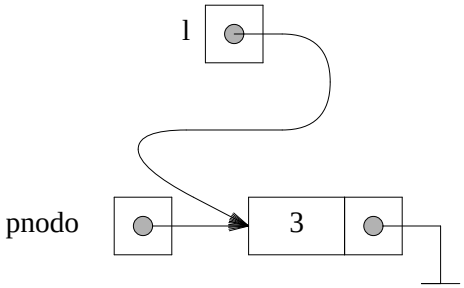
Nótese que no hemos inicializado el nodo. Por tanto, tanto el valor como el puntero al siguiente nodo (contenidos en el nodo) tienen valores aleatorios. Si ejecutamos ahora la segunda y la tercera sentencia del procedimiento tenemos:

```
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:

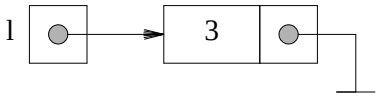
```
7       l = pnodo;
```

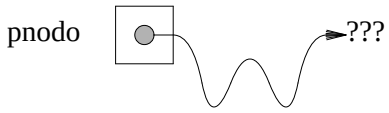


Ahora retornamos de *insertarentero* y la lista *l* se queda perfectamente definida con un nuevo elemento.

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.

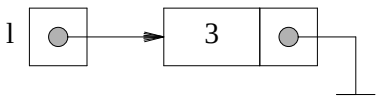
Veamos qué sucede al insertar en número 4 en la lista enlazada según ha quedado tras la inserción anterior. Inicialmente, en la ‘‘`{`’’ del procedimiento, tenemos:

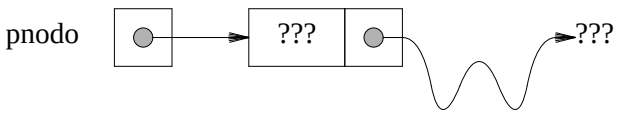




Al ejecutar...

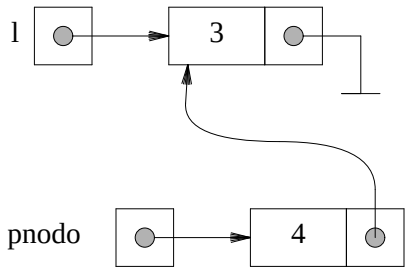
```
4       new(pnodo);
```





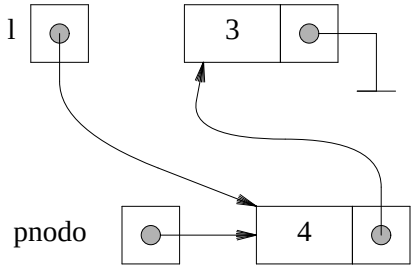
Al ejecutar...

```
5       pnodo^.valor = n;
6       pnodo^.siguiente = l;
```

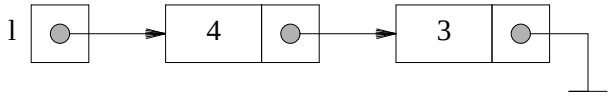


Y por último...

```
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.

Y sí, ¡Si Señor!, el paso de parámetros por referencia no es otra cosa que pedirle al lenguaje que use punteros internamente para referirse a los argumentos. Por eso se llama ‘‘por referencia’’: el lenguaje utiliza un puntero al argumento y ese puntero lo utiliza el código del procedimiento para referirse al argumento. Claro, todo esto lo hace el lenguaje sin que tengamos que hacerlo nosotros (salvo en lenguajes como C y C++, en que tenemos que hacerlo nosotros).

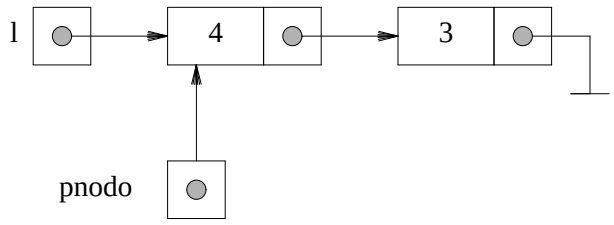
El procedimiento para eliminar un nodo de la cabeza (de la izquierda) de la lista podría ser como sigue:

```
1     procedure eliminarcabeza(ref l: TipoListaEnteros)
2           pnodo: TipoListaEnteros;
3     {
4           if(not eslistaenterosvacia(l)){
5                 pnodo = l;
6                 l = l^.siguiente;
7                 dispose(pnodo);
8           }
9     }
```

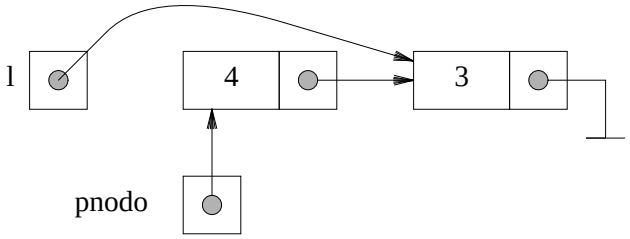
Aquí hay varios detalles importantes. Por un lado, hay que comprobar si la lista está vacía o no. No podemos atravesar el puntero de la lista si esta está vacía. Aunque se supone que nadie debería llamar a *eliminarcabeza* en una lista vacía, es mejor comprobarlo y asegurarse.

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 *nil* si sólo había un nodo), tenemos en *pnodo* el puntero cuyo elemento apuntado hay que liberar. Por ejemplo, al eliminar el primer nodo de la última lista que teníamos se producen los siguientes estados:

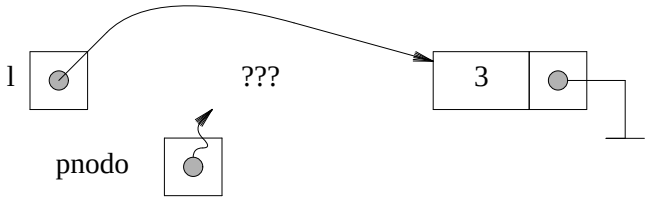
```
5       pnodo = l;
```




```
6       l = l^.siguiente;
```




```
7       dispose(pnodo);
```

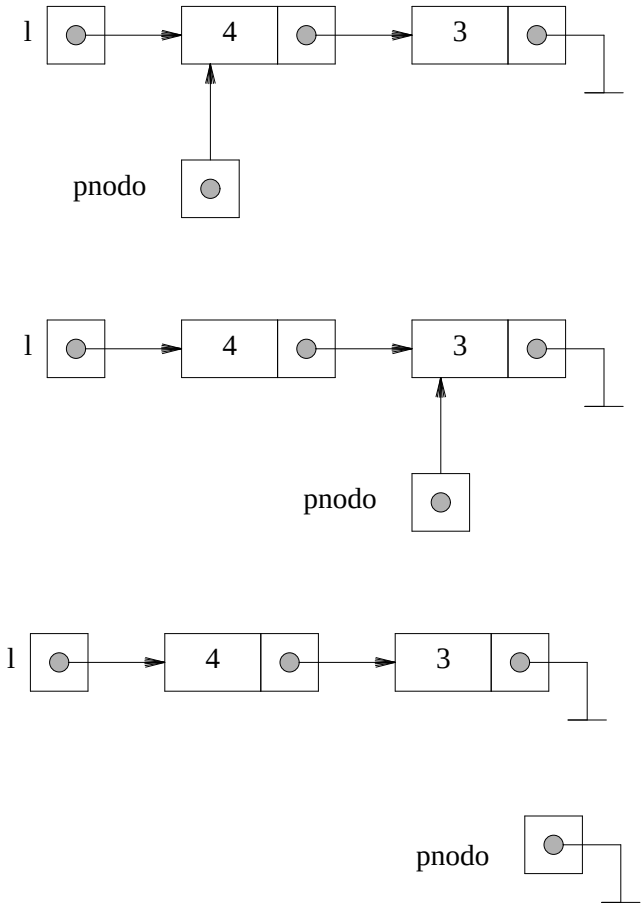


Para continuar con los ejemplos de uso de punteros, el siguiente procedimiento imprime todos los enteros presentes en nuestra lista enlazada.

```
1     procedure escribirlista(l: TipoListaEnteros)
2           pnodo: TipoListaEnteros;
3     {
4           pnodo = l;
5           while(not eslistaenterosvacia(pnodo)){
6                 write(pnodo^.valor);
7                 pnodo = pnodo^.siguiente;
8           }
9     }
```

Utilizamos *pnodo* para ir apuntando sucesivamente a todos los nodos de la lista. En distintas pasadas 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     /*
2      *    Extrae el ultimo elemento.
3      */

5     procedure extraercola(ref l: TipoListaEnteros, ref n: int)
6          pnodo: TipoListaEnteros;
7          panterior: TipoListaEnteros;
8          ultimo: bool;
9     {
10         if(l != nil){
11              pnodo= l;
12              if(l^.siguiente == nil){
13                   n = l^.valor;
14                   l = nil;
15              }else{
16                   panterior = nil;
17                   ultimo = False;
18                   do{
19                        ultimo = pnodo^.siguiente == nil;
20                        if(not ultimo){
21                             panterior = pnodo;
22                             pnodo= pnodo^.siguiente;
23                        }
24                   }while(not ultimo);
25                   panterior^.siguiente = nil;
26                   n = pnodo^.valor;
27              }
28              dispose(pnodo);
29         }
30    }
```

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 *nil* y en el tercero hay que dejar a *nil* el puntero al siguiente 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 *nil*.

Hay formas más compactas de implementar esto pero podrían resultar más complicadas de entender en este punto.

11.8. Invertir la entrada con una pila

Queremos ver si la entrada estándar es un palíndromo, pero sin restricciones respecto a cómo de grande puede ser el texto en la entrada. Como ya sabemos de problemas anteriores un palíndromo es un texto que se lee igual al derecho que al revés.

Una forma de comprobar si un texto es palíndromo 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.

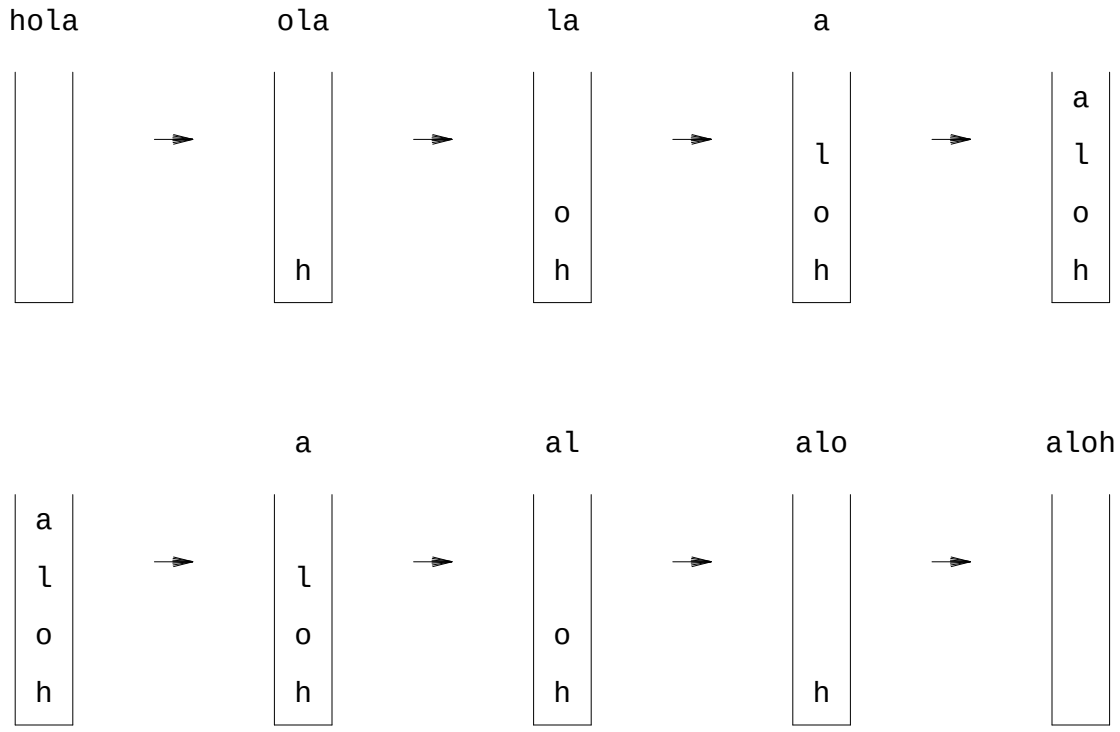


Figura 11.5: *Invertir un texto usando una pila.*

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 como un puntero a un nodo, el cual tiene un carácter y un puntero al resto de la lista.

```
7     types:
8          TipoPilaCar = ^TipoNodoCar;
9          TipoNodoCar = record
10         {
11              c: char;
12              siguiente: TipoPilaCar;
13         };
```

Tradicionalmente se denomina *push* a la operación de insertar algo en una pila y *pop* a la 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     procedure main()
2           pila: TipoPilaCar;
3           c: char;
4     {
5           pila = nuevapila();
6           do{
7                 read(c);
8                 if(eol()){
9                       readeol();
10                }else if(not eof()){
11                      push(pila, c);
12                }
13          }while(not eof());
14
15          while(not esvacia(pila)){
16                pop(pila, c);
17                write(c);
18          }
19    }
—
```

Como mostraba la figura, introducimos todo el texto en una pila 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.

```
24    procedure push(ref pila: TipoPilaCar, c: char)
25          pnodo: TipoPilaCar;
26    {
27          new(pnodo);
28          pnodo^.c = c;
29          pnodo^.siguiente = pila;
30          pila = pnodo;
31    }
```

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.

```
33    procedure pop(ref pila: TipoPilaCar, ref c: char)
34          pnodo: TipoPilaCar;
35    {
36          c = pila^.c;
37          pnodo= pila;
38          pila = pila^.siguiente;
39          dispose(pnodo);
40    }
```

Una mejora que deberíamos hacer es manejar con cuidado los saltos de línea. Cuando leemos, podemos obtener *Eol* si estamos ante un fin de línea. Dado que es un *char*, podemos meterlo en la pila. Si al extraer caracteres de la pila encontramos ese valor, entonces reproducimos el salto de línea. Ahora el programa funciona como esperamos aunque tengamos una entrada con varias líneas:

```
; pick invertir.p
; pam.out
hola
que tal
Control-d
lat euq
aloh
```

Y este es el programa:

invertir.p

```
1     /*
2      *    Invertir la entrada estandar
3      */

5     program invertir;

7     types:
8          TipoPilaCar = ^TipoNodoCar;
9          TipoNodoCar = record
10         {
11              c: char;
12              siguiente: TipoPilaCar;
13         };

15    function nuevapila(): TipoPilaCar
16    {
17         return nil;
18    }

20    function esvacia(pila: TipoPilaCar): bool
21    {
22         return pila == nil;
23    }

25    procedure push(ref pila: TipoPilaCar, c: char)
26         pnodo: TipoPilaCar;
27    {
28         new(pnodo);
29         pnodo^.c = c;
30         pnodo^.siguiente = pila;
31         pila = pnodo;
32    }

34    procedure pop(ref pila: TipoPilaCar, ref c: char)
35         pnodo: TipoPilaCar;
36    {
37         c = pila^.c;
38         pnodo= pila;
39         pila = pila^.siguiente;
40         dispose(pnodo);
41    }
```

```
43    procedure main()
44          pila: TipoPilaCar;
45          c: char;
46    {
47          pila = nuevapila();
48          do{
49                read(c);
50                if(c == Eol){
51                      readeol();
52                }
53                if(c != Eof){
54                      push(pila, c);
55                }
56          }while(not eof());

58          while(not esvacia(pila)){
59                pop(pila, c);
60                if(c == Eol){
61                      writeeol();
62                }else{
63                      write(c);
64                }
65          }
66    }
```

—

11.9. ¿Es la entrada un palíndromo?

Para ver si la entrada (sea lo extensa que sea) es un palíndromo 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           do{
4                 read(c);
5                 if(c == Eol){
6                       readeol();
7                 }
9                 if(c != Eof){
10                      push(pila, c);
11                      insertar(cola, c);
12                }
13          }while(not eof());
```

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 un palíndromo.

```
1          espalindromo = True;
2          while(not espilavacia(pila) and not escolavacia(cola) and espalindromo){
4               pop(pila, c);
5               extraer(cola, c2);
6               espalindromo = c == c2;
7          }
```

Si la entrada es un palíndromo, al final tanto la pila como la cola deberán estar vacías, dado que hemos insertado el mismo número de caracteres en ambas. Pero si no es un palíndromo, el bucle acaba antes de haber extraído y liberado todos los nodos. Por eso necesitaremos implementar procedimientos para liberar la cola y la pila, y llamarlos al final del procedimiento principal.

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.

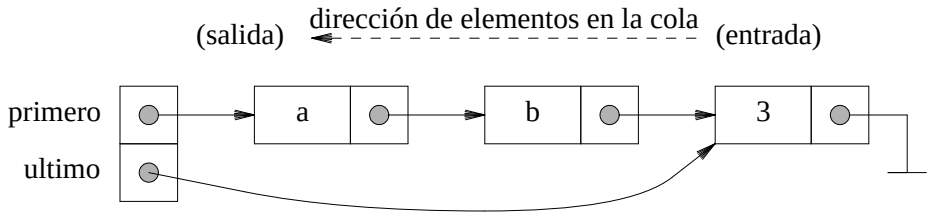


Figura 11.6: *Nuestra implementación para la cola con un puntero al primer nodo y otro al último.*

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 es parecido al 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          TipoPtrCola = ^TipoNodoCola;
2          TipoNodoCola = record
3          {
4               c: char;
5               siguiente: TipoPtrCola;
6          };
7          TipoColaCar = record
8          {
9               primero: TipoPtrCola;
10              ultimo: TipoPtrCola;
11         };
```

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 *nil* (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.

palindromo.p

```
1     /*
2      *    Comprobar si la entrada es un palindromo
3      */

5     program palindromo;
```

```
7     types:
8          TipoPilaCar = ^TipoNodoCar;
9          TipoNodoCar = record
10         {
11              c: char;
12              siguiente: TipoPilaCar;
13         };

15         TipoPtrCola = ^TipoNodoCola;
16         TipoNodoCola = record
17         {
18              c: char;
19              siguiente: TipoPtrCola;
20         };
21         TipoColaCar = record
22         {
23              primero: TipoPtrCola;
24              ultimo: TipoPtrCola;
25         };

27    /*
28     * subprogramas para la pila
29     */

31    function nuevapila(): TipoPilaCar
32    {
33         return nil;
34    }

36    function espilavacia(pila: TipoPilaCar): bool
37    {
38         return pila == nil;
39    }

41    procedure push(ref pila: TipoPilaCar, c: char)
42         pnodo: TipoPilaCar;
43    {
44         new(pnodo);
45         pnodo^.c = c;
46         pnodo^.siguiente = pila;
47         pila = pnodo;
48    }

50    procedure pop(ref pila: TipoPilaCar, ref c: char)
51         pnodo: TipoPilaCar;
52    {
53         c = pila^.c;
54         pnodo= pila;
55         pila = pila^.siguiente;
56         dispose(pnodo);
57    }

59    /*
60     * subprogramas para la cola
61     */
```

```
63   function nuevacola(): TipoColaCar
64        cola: TipoColaCar;
65   {
66        cola.primero = nil;
67        cola.ultimo = nil;
68        return cola;
69   }

71   function escolavacia(cola: TipoColaCar): bool
72   {
73        return cola.primero == nil;
74   }

76   procedure insertar(ref cola: TipoColaCar, c: char)
77        pnodo: TipoPtrCola;
78   {
79        new(pnodo);
80        pnodo^.c = c;
81        pnodo^.siguiente = nil;
82        if(escolavacia(cola)){
83             cola.primero = pnodo;
84        }else{
85             cola.ultimo^.siguiente = pnodo;
86        }
87        cola.ultimo = pnodo;
88   }

90   procedure extraer(ref cola: TipoColaCar, ref c: char)
91        pnodo: TipoPtrCola;
92   {
93        c = cola.primero^.c;
94        pnodo= cola.primero;
95        if(cola.ultimo == cola.primero){
96             cola.ultimo = nil;
97        }
98        cola.primero = cola.primero^.siguiente;
99        dispose(pnodo);
100  }

102  procedure liberarpila(pila: TipoPilaCar)
103       c: char;
104  {
105       while(not espilavacia(pila)){
106            pop(pila, c);
107       }
108  }
109  procedure liberarcola(cola: TipoColaCar)
110       c: char;
111  {
112       while(not escolavacia(cola)){
113            extraer(cola, c);
114       }
115  }
```

```
117  procedure main()
118       pila: TipoPilaCar;
119       cola: TipoColaCar;
120       c: char;
121       c2: char;
122       espalindromo: bool;
123  {
124       pila = nuevapila();
125       cola = nuevacola();
126       do{
127            read(c);
128            if(c == Eol){
129                 readeol();
130            }
131            if(c != Eof){
132                 push(pila, c);
133                 insertar(cola, c);
134            }
135       }while(not eof());

137       espalindromo = True;
138       while(not espilavacia(pila) and not escolavacia(cola) and espalindromo){
139            pop(pila, c);
140            extraer(cola, c2);
141            espalindromo = c == c2;
142       }

144       if(espalindromo){
145            writeln("palindromo");
146       }else{
147            writeln("no palindromo");
148       }

150       liberarpila(pila);
151       liberarcola(cola);
152  }
```

—

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 un palíndromo.

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 esta forma un *string* pequeño usa en realidad un *array* (el del primer nodo) pero uno más largo utiliza una lista y puede crecer.

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. Por cierto, hoy se siguen utilizando para situaciones en que queremos editar y no tenemos gráficos (y en algunas situaciones en que ni siquiera tenemos una pantalla).

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. Nosotros vamos a utilizar directamente la entrada estándar, por el momento.

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.

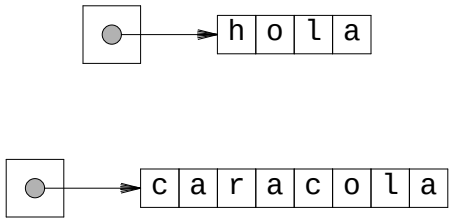


Figura 12.1: *Varios strings dinámicos*

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.

Bueno, hemos contado esto puesto que en muchos lenguajes es la solución más directa. Puedes ver el capítulo 12 de la edición anterior de este libro (escrita utilizando el lenguaje Ada) para ver cómo se podrían implementar este tipo de *strings*.

En Picky la solución mas simple es la que probablemente hubieses imaginado tu sólo: utilizar una lista de caracteres para implementar una palabra, en lugar de un *array* de caracteres. Así que podríamos empezar por ahí:

```
1     types:
2          TipoPalabra = ^TipoNodoCar;
3          TipoNodoCar = record
4          {
5               c: char;
6               siguiente: TipoPalabra;
7          };
```

Durante un tiempo nos vamos a olvidar por completo del editor. Vamos a pensar sólo en programar palabras de longitud variable. Pensando en estas palabras, por ejemplo, en leerlas, seguro que vamos a querer añadir caracteres a una palabra. Como vimos antes, es mas fácil utilizar entonces un puntero al último nodo además de un puntero al primero (como hicimos con la cola). Eso vamos a hacer, nuestra palabra va a ser en realidad justo como una cola de caracteres:

```
1     types:
2          TipoPtrPal = ^TipoNodoPal;
3          TipoNodoPal = record
4          {
5               c: char;
6               siguiente: TipoPtrPal;
7          };
8          TipoPal = record
9          {
10              primero: TipoPtrPal;
11              ultimo: TipoPtrPal;
12         };
```

¡Ah! No lo hemos dicho y no lo volveremos a decir, pero ya hemos compilado el programa dos veces. Una vez tras definir cada tipo de datos (el programa principal ahora mismo tiene sólo la sentencia nula en su cuerpo).

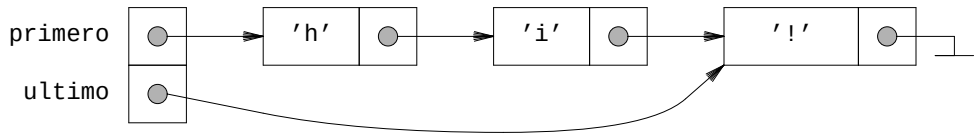


Figura 12.2: *Una palabra es una lista de caracteres con punteros al primero y último.*

¡A divertirse!, podemos empezar a programar *leerpalabra*, *escribirpalabra* y cualquier otra operación que sepamos en este momento que vamos a necesitar. Si se nos olvida alguna, ya la haremos después.

Lo primero podría ser una operación para crear una palabra vacía. Podemos copiar casi tal cual la que teníamos en el caso de la cola de caracteres.

```
1     function nuevapal(): TipoPal
2          pal: TipoPal;
3     {
4          pal.primero = nil;
5          pal.ultimo = nil;
6          return pal;
7     }
```

¿Cómo podemos tener palabras que no sean vacías? ¿Cómo metemos caracteres en una palabra? Lo único que podemos imaginarnos ahora es que, cuando leamos una palabra, iremos añadiendo los caracteres a la palabra conforme los leamos. Vamos pues a hacer un procedimiento *appendcar* que añade (a esto se le suele llamar ‘‘append’’) un caracter a una palabra. Lo bueno es que este procedimiento es justo como el de insertar un carácter en una cola. ¡Ya lo teníamos! Nosotros vamos a añadir caracteres por la derecha (el final) de la cola.

```
1     procedure appendcar(ref pal: TipoPal, c: char)
2           pnodo: TipoPtrPal;
3     {
4           new(pnodo);
5           pnodo^.c = c;
6           pnodo^.siguiente = nil;
7           if(pal.primero == nil){
8                 pal.primero = pnodo;
9           }else{
10                pal.ultimo^.siguiente = pnodo;
11          }
12          pal.ultimo = pnodo;
13    }
```

Primero creamos un nodo con el nuevo carácter. Si la palabra está vacía, hay que hacer que tanto *primero* como *ultimo* apunten al nuevo nodo. En otro caso, basta enlazar el nuevo nodo desde el puntero al siguiente que tenemos en el último nodo (y actualizar cuál es el último nodo).

En realidad, la primera versión que escribimos para *appendcar* hacía esto:

```
1           if(pal.primero == nil){
2                 pal.primero = pnodo;
3                 pal.ultimo = pnodo;
4           }else{
5                 pal.ultimo^.siguiente = pnodo;
6                 pal.ultimo = pnodo;
7           }
```

Luego la miramos y vimos que en realidad podíamos escribir:

```
1           if(pal.primero == nil){
2                 pal.primero = pnodo;
3           }else{
4                 pal.ultimo^.siguiente = pnodo;
5           }
6           pal.ultimo = pnodo;
```

Esa es la idea. Se aplica la técnica del refinamiento progresivo hasta que estamos contentos con el código que tenemos. No hay ningún problema en que la primera versión no esté perfecta. De hecho, casi nunca lo estará.

Ahora podríamos por ejemplo declarar un *array* de caracteres y rellenar una palabra con esos caracteres, para ver si todo va bien. Pero antes de hacer eso, estaría bien poder escribir nuestra palabra.

```
1     procedure writepal(pal: TipoPal)
2           pnodo: TipoPtrPal;
3     {
4           pnodo = pal.primero;
5           while(pnodo != nil){
6                 write(pnodo^.c);
7                 pnodo = pnodo^.siguiente;
8           }
9     }
```

Vamos a probar todo esto. Este es nuestro programa para hacer la prueba, la versión 1 de *e*:

[e.p]

```
1     /*
2      * E, el editor definitivo.
3      */
```

```
5     program e;

7     consts:
8           Prueba = "una prueba";

10    types:
11          TipoPtrPal = ^TipoNodoPal;
12          TipoNodoPal = record
13          {
14                c: char;
15                siguiente: TipoPtrPal;
16          };
17          TipoPal = record
18          {
19                primero: TipoPtrPal;
20                ultimo: TipoPtrPal;
21          };

23    function nuevapal(): TipoPal
24          pal: TipoPal;
25    {
26          pal.primero = nil;
27          pal.ultimo = nil;
28          return pal;
29    }

31    procedure appendcar(ref pal: TipoPal, c: char)
32          pnodo: TipoPtrPal;
33    {
34          new(pnodo);
35          pnodo^.c = c;
36          pnodo^.siguiente = nil;
37          if(pal.primero == nil){
38                pal.primero = pnodo;
39          }else{
40                pal.ultimo^.siguiente = pnodo;
41          }
42          pal.ultimo = pnodo;
43    }

45    procedure writepal(pal: TipoPal)
46          pnodo: TipoPtrPal;
47    {
48          pnodo = pal.primero;
49          while(pnodo != nil){
50                write(pnodo^.c);
51                pnodo = pnodo^.siguiente;
52          }
53    }
```

```
55    procedure main()
56          pal: TipoPal;
57          i: int;
58    {
59          pal = nuevapal();
60          for(i = 0, i < len Prueba){
61                appendcar(pal, Prueba[i]);
62          }
63          writepal(pal);
64          writeeol();
65    }
—
```

Y esto es lo que pasa al ejecutarlo:

```
una prueba
memory leaks:
/usr/prof/pickybook/src/e.p:31(10 times)
```

Naturalmente, perdemos memoria. Nunca hemos liberado la memoria dinámica que hemos pedido para almacenar los caracteres de la palabra. Pero al menos vemos que se escribe justo el texto que corresponde a nuestra prueba. Esto marcha.

Para no perder memoria, vamos a tener que poder destruir palabras. Nos hacemos otra operación:

```
1     procedure disposepal(ref pal: TipoPal)
2           pnodo: TipoPtrPal;
3     {
4           while(pal.primero != nil){
5                 pnodo = pal.primero;
6                 pal.primero = pnodo^.siguiente;
7                 dispose(pnodo);
8           }
9           pal.primero = nil;
10          pal.ultimo = nil;
11    }
```

En esta tenemos cuidado de dejar la palabra como si acabásemos de construir una nueva palabra: vacía. Ponemos tanto *ultimo* como *primero* a *nil*.

Si añadimos una llamada a *disposepal(pal)* justo al final de nuestro programa, y lo volvemos a ejecutar, vemos ahora:

```
una prueba
```

Ya no perdemos memoria.

12.4. Nombres

¿Appendpal? ¿Writepal? ¿Disposepal? Sí. Como muchos lenguajes tienen procedimientos llamados *append* que añaden cosas al final, si usamos *appendcar* eso suena a ‘‘añadir un carácter’’. Igualmente, como muchos lenguajes tienen procedimientos *write* que escriben, *writepal* suena a ‘‘escribir palabra’’. Cuando tengas que inventarte una operación parecida a otras que sean bien conocidas, es mejor utilizar los mismos nombres. Así todo el mundo sabe de qué operaciones se trata, con sólo ver el nombre.

De hecho, vamos a cambiar unos cuantos nombres ahora... ¿Qué tal usar *newpal* en lugar de *nuevapal* y *readpal* en lugar de *leerpal*? No se trata de escribir los nombres en inglés. Se trata de usar nombres que suenen a todo el mundo como operaciones que ya se conocen (como por ejemplo *new* y *read*). Utilizando el editor de texto en el que estamos programando, hemos cambiado

los nombres. A partir de ahora los verás cuando listemos el código.

12.5. Palabras, líneas y textos

Pero sigamos, vamos a tener que poder leer palabras. Haremos lo de costumbre, saltar blancos y leer palabras. El procedimiento *saltarblancos* será como siempre. El procedimiento *leerpal*... ¡Un momento!, ahora es el procedimiento *readpal*. Bueno, pues *readpal* tiene que leer caracteres mientras puedan formar parte de una palabra y añadirlos a la palabra. Como de constumbre, suponemos que se sabe que hay al menos un carácter en la entrada para una palabra antes de llamarlo.

```
1     /*
2      * Hay que saltar blancos y saber que hay una palabra antes de llamarlo.
3      */
4     procedure readpal(ref pal: TipoPal)
5          c: char;
6     {
7          pal = newpal();
8          do{
9               read(c);
10              appendcar(pal, c);
11              peek(c);
12         }while(not eof() and not eol() and not esblanco(c));
13    }
```

Si ahora cambiamos el programa principal para que ejecute esto,

```
1          readpal(pal);
2          writepal(pal);
3          disposepal(pal);
```

podemos ejecutar el programa y ver que escribe la palabra que le damos en la entrada estándar.

Lo siguiente que podríamos hacer ahora es empezar por leer nuestro fichero en una sóla palabra e imprimirlo, para ejercitar nuestro nuevo tipo de datos y sus operaciones. Para hacerlo, podríamos utilizar el típico bucle *while* hasta *eof*, e insertar todos los caracteres que leemos de la entrada en la palabra. Nosotros vamos a seguir adelante.

Realmente siempre se puede mejorar el conjunto de operaciones de un tipo de datos. La clave está en partir de un conjunto razonable y en algún momento decir basta y dejar de mejorarlo. Naturalmente, si luego vemos que nos faltan operaciones, tendremos que implementarlas. Nadie es perfecto.

Queríamos nuestras palabras 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. ¡Pero seamos astutos! Lo mismo que sucedía con las palabras y los caracteres, seguro que también vamos a querer añadir a una línea una nueva palabra. Pues usamos la misma estructura. Ha funcionado bien para una palabra, ¿Por qué no va funcionar para una línea?

```
1          TipoPtrLin = ^TipoNodoLin;
2          TipoNodoLin = record
3          {
4               pal: TipoPal;
5               siguiente: TipoPtrLin;
6          };
7          TipoLin = record
8          {
9               primero: TipoPtrLin;
10              ultimo: TipoPtrLin;
11         };
```

Como verás, es exactamente igual que una palabra, pero aquí mantenemos una lista de palabras y no una lista de caracteres. La figura 12.3 muestra cómo es una línea según la define este tipo de datos. Como verás por la figura, ¡Podemos ignorar por completo cómo están hechas las palabras! Ahora una palabra es un *TipoPal*. 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.

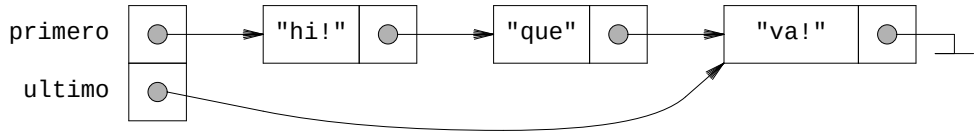


Figura 12.3: *Una línea es una lista de nodos con palabras con punteros al primero y último.*

Si no estás convencido de lo importante que es abstraer, la figura 12.4 muestra la misma línea, pero mostrando realmente cómo están almacenadas las palabras. Sinceramente, nosotros preferimos pensar que una línea es una serie de palabras y olvidarnos de la figura 12.4.

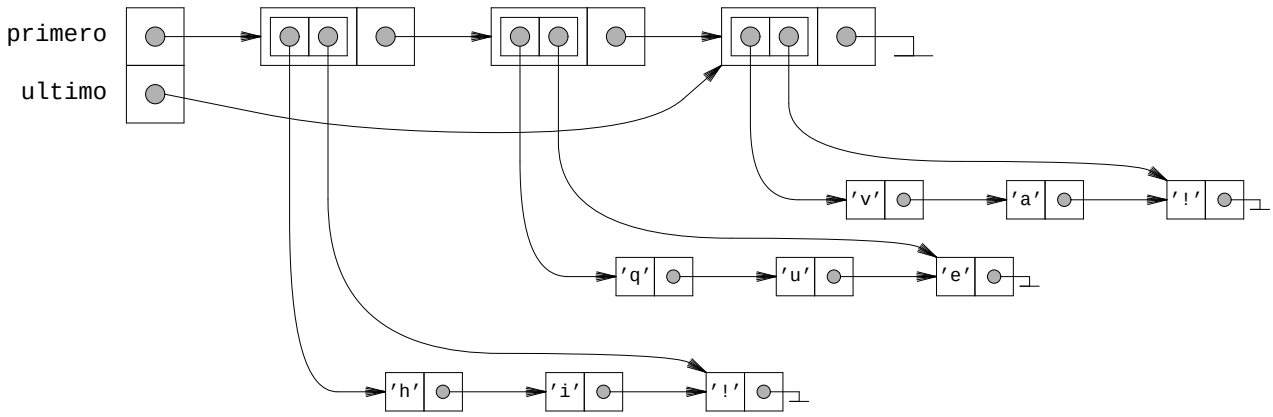


Figura 12.4: *La misma línea, si no abstraemos...*

Ya te imaginarás que para un texto, podemos hacer lo mismo: una lista de líneas. La razón es que también creemos que tendremos que mantener una serie de líneas, sin límite en el número de líneas. Además, vamos a tener que añadir líneas al final de las que ya tenemos. ¡Pues hacemos lo mismo!

```
1          TipoPtrTxt = ^TipoNodoTxt;
2          TipoNodoTxt = record
3          {
4                lin: TipoLin;
5                siguiente: TipoPtrTxt;
6          };
7          TipoTxt = record
8          {
9                primero: TipoPtrTxt;
10               ultimo: TipoPtrTxt;
11         };
```

Pero volvamos a las líneas. Es pronto para el 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. Las operaciones van a ser exactamente iguales a las que teníamos para las palabras, salvo por que ahora utilizamos palabras como elementos en lugar de caracteres.

Vamos a ver por ejemplo *appendpal*, similar a *appendcar*, pero ocupándose de añadir una palabra a una línea (y no un carácter a una palabra):

```
1     procedure appendpal(ref lin: TipoLin, pal: TipoPal)
2          pnodo: TipoPtrLin;
3     {
4          new(pnodo);
5          pnodo^.pal = pal;     /* Esta mal */
6          pnodo^.siguiente = nil;
7          if(lin.primero == nil){
8                lin.primero = pnodo;
9          }else{
10               lin.ultimo^.siguiente = pnodo;
11         }
12         lin.ultimo = pnodo;
13    }
```

El código es como esperábamos, si miramos *appendcar*. Pero fíjate en la línea 5. Esta línea está mal. ¿Qué pasará cuando se produzca la asignación? Bueno, pues que el record *pnodo^.pal* se copiará, elemento a elemento, del record *pal*. ¿Es eso lo que queremos?

Si hacemos esa asignación, ambos records compartirán los nodos de la lista de caracteres que forman la palabra. Mira la figura 12.5. ¿Y qué pasará luego si alguien ejecuta *dispose(pal)*?

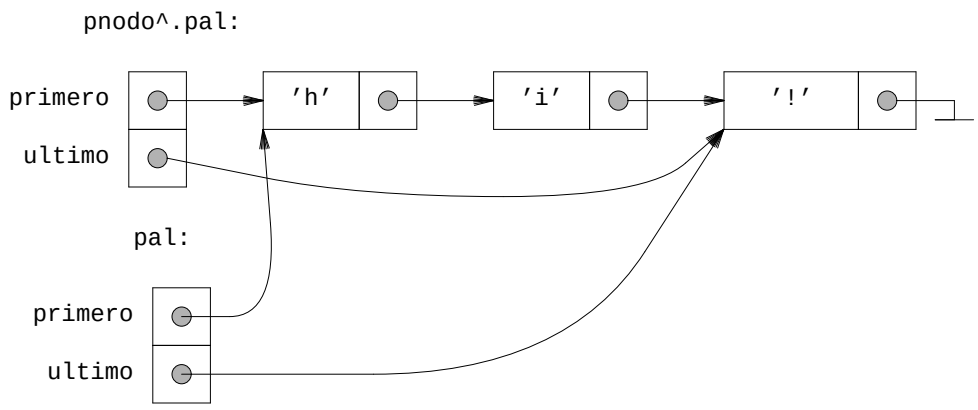


Figura 12.5: *Las palabras son estructuras dinámicas y si las asignamos directamente...*

Siempre que tenemos una estructura de datos que usa punteros, tenemos que evitar copiarla usando directamente la asignación. Tenemos que hacer un procedimiendo *copiar* y utilizarlo. Dicho procedimiento debe copiar todos los nodos, no sólo los punteros del comienzo de la lista. Una vez tengamos ese procedimiento, podemos cambiar *appendpal* para que sea como sigue:

```
1     procedure appendpal(ref lin: TipoLin, pal: TipoPal)
2          pnodo: TipoPtrLin;
3     {
4          new(pnodo);
5          copiarpal(pnodo^.pal, pal);
6          pnodo^.siguiente = nil;
7          if(lin.primero == nil){
8               lin.primero = pnodo;
9          }else{
10              lin.ultimo^.siguiente = pnodo;
11         }
12         lin.ultimo = pnodo;
13    }
```

Cuando implementamos *disposelin*, el procedimiento análogo a *disposepal*, pasa lo mismo. Hemos de liberar cada palabra (que usa memoria dinámica) antes de liberar cada nodo de la lista.

```
1     procedure disposelin(ref lin: TipoLin)
2          pnodo: TipoPtrLin;
3     {
4          while(lin.primero != nil){
5               pnodo = lin.primero;
6               lin.primero = pnodo^.siguiente;
7               disposepal(pnodo^.pal);
8               dispose(pnodo);
9          }
10         lin.primero = nil;
11         lin.ultimo = nil;
12    }
```

Igualmente, *writelin* no puede utilizar *write* para escribir cada palabra de la línea. Debe llamar en su lugar a *writepal*. ¡Ah! y además debería hacer un *writeeol* tras escribir todas las palabras. Al fin y al cabo está escribiendo una línea, y las líneas terminan en un fin de línea.

¿Cómo copiamos una palabra en otra? Podemos recorrer los caracteres de una palabra y añadirlos a la palabra destino. Pero claro, antes de eso hay que inicializar la palabra destino. Si miras *appendpal*, verás que *copiarpal* se está utilizando para inicializar *pnodoˆ.pal*, con lo que estamos suponiendo que *copiarpal* no debe depender del valor inicial de la palabra destino. Lo mismo que hace la asignación. Y por cierto, el orden de los argumentos de *copiarpal* también es el orden que se usaría en una asignación. Así no hay dudas. Este es el procedimiento:

```
1     procedure copiarpal(ref pal: TipoPal, orig: TipoPal)
2          pnodo: TipoPtrPal;
3     {
4          pal = newpal();
5          pnodo = orig.primero;
6          while(pnodo != nil){
7               appendcar(pal, pnodo^.c);
8               pnodo = pnodo^.siguiente;
9          }
10    }
```

Lo podemos probar utilizando un programa principal que lee una palabra, la copia en otra y escribe esta última:

```
1          readpal(pal);
2          copiarpal(pal2, pal);
3          writepal(pal2);
4          disposepal(pal);
5          disposepal(pal2);
```

En este punto tal vez fuese mejor hacer un programa que leyese todo el texto del fichero en una única línea, palabra por palabra. Nosotros seguiremos adelante.

12.6. Palabras y blancos

Si has hecho el programa que acabamos de sugerir, tal vez te lleves una sorpresa. Antes de leer cada palabra que añadimos a la línea, llamando a *readpal*, hay que *saltarblancos*. Luego la línea no contiene el espacio en blanco que había en la entrada. Eso no nos sirve para un editor. Podemos cambiar nuestro *saltarblancos* por un *readblancos* para que ahora lea una variable de *TipoPal* que contiene los blancos que se han leído.

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 *TipoPal* también para almacenar los blancos. Este procedimiento salta blancos de la entrada, leyéndolos en un *TipoPal*:

```
1     procedure readblancos(ref pal: TipoPal)
2          c: char;
3     {
4          pal = newpal();
5          do{
6               peek(c);
7               if(esblanco(c)){
8                    read(c);
9                    appendcar(pal, c);
10              }
11         }while(not eof() and not eol() and esblanco(c));
12    }
```

Como queremos leer blancos dentro de una línea, hemos tenido que dejar de leer también en el fin de línea, cosa que no hacía *saltarblancos*.

Si pensamos un poco, tanto *readblancos* como *readpal* están haciendo lo mismo; aunque *readblancos* está mejor hecho. La diferencia es que uno sigue leyendo mientras *esblanco* y el otro hace justo lo contrario. Podemos inventarnos un nuevo tipo:

```
1          TipoQue = (Blancos, NoBlancos); /* para readpal() */
```

Y hacer que nuestro *leerblancos* sea también nuestro *readpal*:

```
1     procedure readpal(ref pal: TipoPal, que: TipoQue)
2          c: char;
3          esnuestro: bool;
4     {
5          pal = newpal();
6          do{
7               peek(c);
8               esnuestro = ((esblanco(c) and que == Blancos) or
9                    (not esblanco(c) and que == NoBlancos));
10              if(esnuestro){
11                   read(c);
12                   appendcar(pal, c);
13              }
14         }while(not eof() and not eol() and esnuestro);
15    }
```

Naturalmente hemos borrado el antiguo *readpal*, que es equivalente a llamar al nuevo pasando *NoBlancos* como argumento para el parámetro *que*.

Y ahora estamos en situación de leer una línea de texto. Podemos entrar en un bucle leyendo blancos y palabras alternativamente. Algo como...

```
1     procedure readlin(ref lin: TipoLin)
2           pal: TipoPal;
3           c: char;
4     {
5           lin = newlin();
6           do{
7                 peek(c);
8                 if(c != Eol and c != Eof){
9                       if(esblanco(c)){
10                            readpal(pal, Blancos);
11                      }else{
12                            readpal(pal, NoBlancos);
13                      }
14                      appendpal(lin, pal);
15                      disposepal(pal);
16                }
17          }while(not eof() and not eol());
18          if(eol()){
19                readeol();
20          }
21    }
```

Lo que hacemos es mirar con *peek* y, si hay un blanco, llamamos a *readpal* para leer blancos (esto es, ‘‘*readblancos*’’); si hay otra cosa, entonces ha de ser una palabra y llamamos *readpal* para leer no-blancos. Naturalmente, evitando el fin de fichero y el fin de línea. Cuando hemos terminado, tenemos que saltar el fin de línea (si no hemos encontrado antes el fin de fichero), eso es lo que se espera de una función que lee una línea. Y claro, puede que tengamos líneas vacías cuando encontramos una línea en blanco.

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 *appendpal* 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.

12.7. Textos

Una vez tenemos líneas podemos considerar un texto completo. Para nosotros el texto de un fichero será una serie de líneas. Ya teníamos el tipo de datos programado, cuando sospechamos que sería similar al de las líneas. Por el momento, podemos programar *newtxt*, *appendlin*, *writetxt* y *disposetxt* exactamente como hicimos con *newlin*, *appendpal*, *writelin* y *disposelin*. Solo que en lugar de palabras y líneas, ahora tenemos líneas y texto. Claro, vamos a necesitar un *copiarlin* por la misma razón que tenemos un *copiarpal*. Pero se programa igual. ¿Y cómo hacemos *readtxt*? Basta leer líneas hasta fin de fichero. Por no complicar mas las cosas, suponemos que es aceptable que si leemos un fichero vacío, nos quede un texto que tiene una única línea vacía. Si queremos evitar tal cosa, deberíamos utilizar de nuevo *peek*.

```
1     procedure readtxt(ref txt: TipoTxt)
2          lin: TipoLin;
3     {
4          txt = newtxt();
5          do{
6               readlin(lin);
7               appendlin(txt, lin);
8               disposelin(lin);
9          }while(not eof());
10    }
```

Y si ahora compilamos nuestro editor y lo ejecutamos, usando su propio código fuente como entrada, podemos ver lo que sigue:

[e.p]

```
1     /*
2      * E, el editor definitivo.
3      */

5     program e;

7     consts:
8          Prueba = "una prueba";

10    types:
11         TipoQue = (Blancos, NoBlancos); /* para readpal() */

13         /*
14          * Palabras
15          */
16         TipoPtrPal = ^TipoNodoPal;
17         TipoNodoPal = record
18         {
19              c: char;
20              siguiente: TipoPtrPal;
21         };
22         TipoPal = record
23         {
24              primero: TipoPtrPal;
25              ultimo: TipoPtrPal;
26         };

28         /*
29          * Lineas
30          */
31         TipoPtrLin = ^TipoNodoLin;
32         TipoNodoLin = record
33         {
34              pal: TipoPal;
35              siguiente: TipoPtrLin;
36         };
37         TipoLin = record
38         {
39              primero: TipoPtrLin;
40              ultimo: TipoPtrLin;
41         };
```

```
43          /*
44           * Texto
45           */
46          TipoPtrTxt = ^TipoNodoTxt;
47          TipoNodoTxt = record
48          {
49               lin: TipoLin;
50               siguiente: TipoPtrTxt;
51          };
52          TipoTxt = record
53          {
54               primero: TipoPtrTxt;
55               ultimo: TipoPtrTxt;
56          };

58     function newpal(): TipoPal
59          pal: TipoPal;
60     {
61          pal.primero = nil;
62          pal.ultimo = nil;
63          return pal;
64     }

66     procedure appendcar(ref pal: TipoPal, c: char)
67          pnodo: TipoPtrPal;
68     {
69          new(pnodo);
70          pnodo^.c = c;
71          pnodo^.siguiente = nil;
72          if(pal.primero == nil){
73               pal.primero = pnodo;
74          }else{
75               pal.ultimo^.siguiente = pnodo;
76          }
77          pal.ultimo = pnodo;
78     }

80     procedure writepal(pal: TipoPal)
81          pnodo: TipoPtrPal;
82     {
83          pnodo = pal.primero;
84          while(pnodo != nil){
85               write(pnodo^.c);
86               pnodo = pnodo^.siguiente;
87          }
88     }

90     procedure disposepal(ref pal: TipoPal)
91          pnodo: TipoPtrPal;
92     {
93          while(pal.primero != nil){
94               pnodo = pal.primero;
95               pal.primero = pnodo^.siguiente;
96               dispose(pnodo);
97          }
98          pal.primero = nil;
99          pal.ultimo = nil;
100    }
```

```
102  procedure copiarpal(ref pal: TipoPal, orig: TipoPal)
103       pnodo: TipoPtrPal;
104  {
105       pal = newpal();
106       pnodo = orig.primero;
107       while(pnodo != nil){
108            appendcar(pal, pnodo^.c);
109            pnodo = pnodo^.siguiente;
110       }
111  }

113  function newlin(): TipoLin
114       lin: TipoLin;
115  {
116       lin.primero = nil;
117       lin.ultimo = nil;
118       return lin;
119  }

121  procedure appendpal(ref lin: TipoLin, pal: TipoPal)
122       pnodo: TipoPtrLin;
123  {
124       new(pnodo);
125       copiarpal(pnodo^.pal, pal);
126       pnodo^.siguiente = nil;
127       if(lin.primero == nil){
128            lin.primero = pnodo;
129       }else{
130            lin.ultimo^.siguiente = pnodo;
131       }
132       lin.ultimo = pnodo;
133  }

135  procedure writelin(lin: TipoLin)
136       pnodo: TipoPtrLin;
137  {
138       pnodo = lin.primero;
139       while(pnodo != nil){
140            writepal(pnodo^.pal);
141            pnodo = pnodo^.siguiente;
142       }
143       writeeol();
144  }

146  procedure disposelin(ref lin: TipoLin)
147       pnodo: TipoPtrLin;
148  {
149       while(lin.primero != nil){
150            pnodo = lin.primero;
151            lin.primero = pnodo^.siguiente;
152            disposepal(pnodo^.pal);
153            dispose(pnodo);
154       }
155       lin.primero = nil;
156       lin.ultimo = nil;
157  }
```

```
159  procedure copiarlin(ref lin: TipoLin, orig: TipoLin)
160       pnodo: TipoPtrLin;
161  {
162       lin = newlin();
163       pnodo = orig.primero;
164       while(pnodo != nil){
165            appendpal(lin, pnodo^.pal);
166            pnodo = pnodo^.siguiente;
167       }
168  }

170  function newtxt(): TipoTxt
171       txt: TipoTxt;
172  {
173       txt.primero = nil;
174       txt.ultimo = nil;
175       return txt;
176  }

178  procedure appendlin(ref txt: TipoTxt, lin: TipoLin)
179       pnodo: TipoPtrTxt;
180  {
181       new(pnodo);
182       copiarlin(pnodo^.lin, lin);
183       pnodo^.siguiente = nil;
184       if(txt.primero == nil){
185            txt.primero = pnodo;
186       }else{
187            txt.ultimo^.siguiente = pnodo;
188       }
189       txt.ultimo = pnodo;
190  }

192  procedure writetxt(txt: TipoTxt)
193       pnodo: TipoPtrTxt;
194  {
195       pnodo = txt.primero;
196       while(pnodo != nil){
197            writelin(pnodo^.lin);
198            pnodo = pnodo^.siguiente;
199       }
200  }

202  procedure disposetxt(ref txt: TipoTxt)
203       pnodo: TipoPtrTxt;
204  {
205       while(txt.primero != nil){
206            pnodo = txt.primero;
207            txt.primero = pnodo^.siguiente;
208            disposelin(pnodo^.lin);
209            dispose(pnodo);
210       }
211       txt.primero = nil;
212       txt.ultimo = nil;
213  }
```

```
function esblanco(c: char): bool
{
     return c == ' ' or c == Tab;
}

procedure readpal(ref pal: TipoPal, que: TipoQue)
     c: char;
     esnuestro: bool;
{
     pal = newpal();
     do{
          peek(c);
          esnuestro = ((esblanco(c) and que == Blancos) or
               (not esblanco(c) and que == NoBlancos));
          if(esnuestro){
               read(c);
               appendcar(pal, c);
          }
     }while(not eof() and not eol() and esnuestro);
}


procedure readlin(ref lin: TipoLin)
     pal: TipoPal;
     c: char;
{
     lin = newlin();
     do{
          peek(c);
          if(c != Eol and c != Eof){
               if(esblanco(c)){
                    readpal(pal, Blancos);
               }else{
                    readpal(pal, NoBlancos);
               }
               appendpal(lin, pal);
               disposepal(pal);
          }
     }while(not eof() and not eol());
     if(not eof()){
          readeol();
     }
}

procedure readtxt(ref txt: TipoTxt)
     lin: TipoLin;
{
     txt = newtxt();
     do{
          readlin(lin);
          appendlin(txt, lin);
          disposelin(lin);
     }while(not eof());
}
```

```
270   procedure main()
271         txt: TipoTxt;
272   {
273
274         readtxt(txt);
275         writetxt(txt);
276         disposetxt(txt);
277   }
—
```

¡Alto! en la prueba que hemos hecho, *readtxt* ha escrito una línea más de las que debería. Una en blanco al final del todo. Pensándolo, *readlin* devuelve una línea vacía si se le llama en una situación de fin de fichero y, claro, si acabamos de leer la última línea, aún no nos hemos encontrado con el fin de fichero. Para solucionar esto, tenemos que cambiar *readtxt* para que sea como sigue:

```
1     procedure readtxt(ref txt: TipoTxt)
2           lin: TipoLin;
3           c: char;
4     {
5           txt = newtxt();
6           do{
7                 readlin(lin);
8                 appendlin(txt, lin);
9                 disposelin(lin);
10                peek(c);   /* hay eof? */
11          }while(not eof());
12    }
```

¿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 *writeln* o llamando a *stack* en una zona concreta de código (el nuevo que hemos añadido), y pasar rápido a otra cosa.

12.8. Comandos y records con variantes

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). ¡Pero bueno! ¡Si es casi como la calculadora!

Estos podrían ser los códigos de nuestros comandos:

```
1           TipoCodigo = (Insertar, Borrar, Imprimir, Cambiar, Editar,
2                       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 erróneo.

En cuanto a la aridad de un comando, los tenemos sin argumentos, con uno y con dos argumentos. En realidad, tenemos comandos que usan como argumentos:

- ninguno,
- un rango de números (con ninguno, uno o dos números),
- una palabra,
- dos 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). Y cuando un comando que acepta números de línea no reciba ninguno, le daremos como rango desde la primera hasta la última línea. Cuando leamos comandos ya ajustaremos el rango para que todo cuadre.

Aunque en realidad ‘‘aridad’’ significa número de argumentos, vamos a pervertir un poco su significado y vamos a hacer que en nuestro caso nos diga además qué tipo de argumentos tenemos:

```
1          TipoAridad = (NoArgs, Rango, Palabra, Palabras);
```

Y este es el tipo de datos para un comando:

```
1          TipoRango = record
2          {
3               primero: int;
4               ultimo: int;
5          };

7          TipoCmd = record
8          {
9               codigo: TipoCodigo;
10              aridad: TipoAridad;
11              switch(aridad){
12              case Rango:
13                   rango: TipoRango;     /* se usa solo si aridad es Rango. */
14              case Palabra, Palabras:
15                   pal1: TipoPal;        /* solo si aridad es Palabra o Palabras. */
16              case Palabras:
17                   pal2: TipoPal;        /* solo si aridad es Palabras. */
18              }
19         };
```

El *record* que hemos utilizado para declarar *TipoCmd* es conocido como un **record con variantes** (en otros lenguajes hay algo parecido que se conoce como *union*). En un *record* con variantes tenemos una parte del *record* que es siempre igual y una parte del *record* que varía en función del valor de un campo.

Por ejemplo, en todos nuestros *TipoCmds* usaremos siempre los campos *codigo* y *aridad*. Esta sería la parte fija del record. Ahora bien, dependiendo del valor de *aridad*, sólo vamos a usar alguno de los campos *rango*, *pal1* y *pal2*, incluso puede que ninguno. La idea del *record* con variantes, es que todos estos campos están o no presentes en el record dependiendo del valor del campo *aridad*. Esta sería la parte del *record* que varía.

La única ventaja de un *record* con variantes es que ahorramos algo de memoria (puesto que no todos los campos están presentes) y que el compilador nos puede avisar si por error utilizamos un campo que no deberíamos utilizar. No obstante, dan lugar a tantos errores que mucha gente no los utiliza. Piensa que *rango* y *pal1* ocuparán posiblemente la misma zona en la memoria. Si por error cambiamos *rango* y el compilador no lo detecta, tal vez estaríamos cambiando *pal1*.

Aunque en Picky el compilador no ayuda mucho al detectar errores en el acceso a campos variantes, por el momento, los hemos utilizado sólo para que los conozcas. Si prefieres no usarlos, puedes utilizar simplemente:

```
1          TipoCmd = record
2          {
3               codigo: TipoCodigo;
4               aridad: TipoAridad;
5               rango: TipoRango;    /* se usa solo si aridad es Rango. */
6               pal1: TipoPal;       /* solo si aridad es Palabra o Palabras. */
7               pal2: TipoPal;       /* solo si aridad es Palabras. */
8          };
```

Para leer un comando tenemos que leer primero el carácter que identifica el comando y, luego, leer los argumentos en función del comando en cuestión. Es posible que tengamos algún error al leer el código o los argumentos. Por el momento, este podría ser un candidato a *readcmd*:

```
1     procedure readcmd(ref cmd: TipoCmd)
2          ok: bool;
3     {
4          readcodigo(cmd.codigo);
5          if(cmd.codigo != Error){
6               ok = True;
7               switch(cmd.codigo){
8               case Insertar, Borrar, Imprimir:
9                    readrangoargs(cmd.rango, ok);
10              case Cambiar:
11                   readpalsarg(cmd.pal1, cmd.pal2, ok);
12              case Editar, Guardar:
13                   readpalarg(cmd.pal1, ok);
14              }
15              if(not ok){
16                   cmd.codigo = Error;
17              }
18         }
19    }
```

La idea es que los procedimientos que leen los argumentos devuelven en *ok* una indicación de si han podido leer los argumentos correctamente o no. Suponemos que cuando llamamos a *readcmd* sabemos que no estamos ante fin de fichero ni ante un fin de línea, por lo que al menos hay un carácter que leer. Eso sí, mientras leemos un comando puede pasar cualquier cosa.

Por lo tanto, el procedimiento *readcodigo* puede suponer que tiene el carácter del código:

```
1     procedure readcodigo(ref codigo: TipoCodigo)
2          c: char;
3     {
4          read(c);
5          switch(c){
6          case 'e':
7               codigo = Editar;
8          case 'w':
9               codigo = Guardar;
10         case 'f':
11              codigo = Cual;
```

```
12          case 'x':
13               codigo = Listar;
14          case 'd':
15               codigo = Borrar;
16          case 'i':
17               codigo = Insertar;
18          case 'p':
19               codigo = Imprimir;
20          case 's':
21               codigo = Cambiar;
22          default:
23               codigo = Error;
24          }
25     }
```

Para los comandos *Editar* y *Guardar* vamos a llamar a *readpalarg*, que lee una palabra como argumento. Este procedimiento debe tener ya cuidado con el fin de fichero y el fin de línea.

```
1      procedure readpalarg(ref pal: TipoPal, ref ok: bool)
2      {
3           ok = False;
4           saltarblancos();
5           ok = not eof() and not eol();
6           if(ok){
7                readpal(pal);
8           }
9      }
```

Aunque hay un pequeño problema, como tenemos un comando por línea, deberíamos leer el resto de la línea tras leer los argumentos. Podemos conseguir tal cosa con un procedimiento *saltarlinea* que lea todo lo que tengamos hasta el próximo fin de línea, incluyendolo.

```
1      procedure saltarlinea()
2           c: char;
3      {
4           while(not eol() and not eof()){
5                read(c);
6           }
7           if(eol()){
8                readeol();
9           }
10     }
```

Y añadimos una llamada a *saltarlinea* justo al final de *readcmd.*

Para leer los argumentos del comando *Cambiar*, que era de la forma ‘‘`s/pal/otra/`’’ vamos a tener que leer una palabra deteniéndonos en el carácter ‘‘`/`’’. Eso es un problema, nuestro *readpal* no nos sirve. Podemos leer carácter a carácter. Por ejemplo, si acabamos de leer la ‘‘`s`’’ con el código del comando, podríamos hacer algo como:

```
1          ok = not eof() and not eol();
2          if(ok){
3               read(separador);
4               ok = not eof() and not eol();
5          }
6          newpal(p1);
7          while(ok){
8               peek(c);
9               ok = not eof() and not eol();
10              if(ok and c != separador){
11                   read(c);
12                   appencar(p1, c);
13              }
14         }
```

Esto lee el primer caracter que sigue, tomándolo como un separador y leyendo en *p1* todos los caracteres que encontremos hasta *Eof, Eol* o dicho separador. La variable *ok* queda a *True* cuando lo hemos conseguido.

Pero tiene un error, si llegamos a un separador, no leemos y *ok* sigue a *True*. Tenemos un bucle infinito. En lugar de eso, es mejor hacer esto:

```
1               ok = not eof() and not eol();
2               if(ok){
3                    read(separador);
4                    ok = not eof() and not eol();
5               }
6               newpal(p1);
7               do{
8                    peek(c);
9                    ok = not eof() and not eol();
10                   if(ok and c != separador){
11                        read(c);
12                        appendcar(pals[i], c);
13                   }
14              }while(ok and c != separador);
```

Y ahora deberíamos repetir de nuevo lo mismo para leer la segunda palabra de los argumentos: leer el separador y leer lo que tengamos hasta dicho separador. Mejor utilizar entonces un *array* de palabras

```
1          TipoPalsArg = array[0..1] of TipoPal;
```

y un bucle *for*:

```
1     procedure readpalsarg(ref p1: TipoPal, ref p2: TipoPal, ref ok: bool)
2          separador: char;
3          c: char;
4          i: int;
5          pals: TipoPalsArg;
6     {
```

```
7          separador = '/';          /* no se usa este valor en realidad */
8          for(i = 0, i < len pals){
9               ok = not eof() and not eol();
10              if(ok){
11                   read(separador);
12                   ok = not eof() and not eol();
13              }
14              pals[i] = newpal();
15              do{
16                   peek(c);
17                   ok = not eof() and not eol();
18                   if(ok and c != separador){
19                        read(c);
20                        appendcar(pals[i], c);
21                   }
22              }while(ok and c != separador);
23         }
24         if(ok){
25              read(separador);
26         }
27         p1 = pals[0];
28         p2 = pals[1];
29    }
```

Si lo miras detenidamente verás que *p1* y *p2* siembre resultan correctamente inicializados, pase lo que pase. Además, como utilizamos el *array* sólo para no repetir el código dos veces, no necesitamos copiar *pals[0]* y *pals[1]* a *p1* y *p2* usando *copiarpal*. Nuestro *array* va a dejar de existir en cuanto termine el procedimiento.

Aún así... ¡Un momento! El hecho de que hayamos necesitado declarar un *array* auxiliar se debe a que en realidad hemos metido la pata con la declaración de *TipoCmd*. Si hay comandos que tienen una o dos palabras, o uno o dos enteros, como argumentos, deberían haber utilizado un *array* en lugar de un *record.* En lugar de seguir, es mucho mejor arreglar este problema ahora. Vamos a cambiar el código para que este sea el nuevo tipo que define un comando, y sus argumentos:

```
1          TipoCodigo = (Insertar, Borrar, Imprimir, Cambiar, Editar,
2                        Guardar, Cual, Listar, Error);
3          TipoAridad = (NoArgs, Rango, Palabra, Palabras);

5          TipoArgNum = array[0..MaxNumArg-1] of int;
6          TipoArgPal = array[0..MaxNumArg-1] of TipoPal;

8          TipoCmd = record
9          {
10              codigo: TipoCodigo;
11              aridad: TipoAridad;
12              switch(aridad){
13              case Rango:
14                   num: TipoArgNum;          /* se usa solo si aridad es Rango. */
15              case Palabra, Palabras:
16                   pal: TipoArgPal;          /* solo si aridad es Palabra o Palabras. */
17              }
18         };
```

Naturalmente hemos definido también la constante:

```
1          MaxNumArg = 2;  /* numero maximo de argumentos en un comando */
```

Nos falta *readrangoargs*. Este procedimiento debe leer cero, uno o dos números. Podemos simplemente llamar a *leerpal* las veces necesarias y convertir las palabras en números.

```
1     procedure readrangoargs(ref arg: TipoArgNum, ref ok: bool)
2          pal: TipoPal;
3          i: int;
4     {
5          arg[0] = Ninguno;
6          arg[1] = Ninguno;
7          ok = True;
8          for(i = 0, i < len arg){
9               saltarblancos();
10              if(not eof() and not eol()){
11                   readpal(pal, NoBlancos);
12                   convpalnat(pal, arg[i], ok);
13                   disposepal(pal);
14              }
15         }
16    }
```

Hemos definido *Ninguno* como

```
1     consts:
2          Ninguno = -1;   /* para numeros de linea en TipoArgNum */
```

para inicializar los dos números del rango. Si no habia ningún número en la entrada, *rango.primero* y *rango.ultimo* quedan a *Ninguno*, y *ok* queda a *True.* Si había un número, entonces *rango.ultimo* queda a *Ninguno*. Suponemos que *convpalnat* convierte una palabra en un natural, y nos dirá si tenía aspecto de entero o no y qué valor tenía. Vamos a hacer *convpalnat* de tal forma que sólo considere que son correctos valores mayores o iguales a cero, así que esto cuadra con la definición de *Ninguno*.

```
1     procedure convpalnat(ref pal: TipoPal, ref val: int, ref ok: bool)
2          c: char;
3          i: int;
4     {
5          i = 0;
6          val = 0;
7          ok = True;
8          while(ok and i < lenpal(pal)){
9               c = palcar(pal, i);
10              if(not esdigito(c)){
11                   ok = False;
12              }else{
13                   val = val * 10;
14                   val = val + (int(c) - int('0'));
15                   i = i + 1;
16              }
17         }
18    }
```

Nos hemos inventado *lenpal*, que nos dice qué longitud tiene una palabra, y *getcar*, devuelve el n-ésimo carácter de la palabra. También una función *esdigito* que devuelve *True* is un carácter corresponde a un dígito. Puedes ver los procedimientos auxiliares en el listado final del programa, despúes en este capítulo.

¡Pues ya tenemos comandos! Al menos, ya podemos leerlos. Ahora podemos escribir un programa principal que lea comandos y los ejecute.

12.9. Ejecutando comandos

Hasta ahora hemos estado utilizando siempre la entrada estándar. Ahora vamos a empezar por hacer que el editor lea comandos de la entrada estándar, pero lea el texto del fichero indicado por el usuario. Eso quiere decir que los procedimientos que tenemos para leer deben utilizar un parámetro *fich* de tipo *file* que les indique de dónde deben leer. También quiere decir que dichos procedimientos han de utilizar *fread* en lugar de *read*, *feof* en lugar de *eof*, etc. Dado que vamos a tener que guardar también el fichero, cuando nos lo indique el usuario, pasa lo mismo con los procedimientos que escriben. Por ejemplo, este es el nuevo *writepal*.

```
1     procedure writepal(ref fich: file, pal: TipoPal)
2          pnodo: TipoPtrPal;
3     {
4          pnodo = pal.primero;
5          while(pnodo != nil){
6               fwrite(fich, pnodo^.c);
7               pnodo = pnodo^.siguiente;
8          }
9     }
```

Puedes ver el resto de los procedimientos ya cambiados en el listado final del programa, más adelante. Y antes de seguir, ¡Un aviso!: este tipo de cambios es delicado. Si se nos olvida cambiar un *peek* for un *fpeek* puede que cuando ejecutemos nuestro programa veamos que se ‘‘queda colgado’’ en algún punto. Seguramente esté leyendo de la entrada estándar dentro de *peek.* Así que es mejor hacer este tipo de cambios *muy despacito* y cuando se esté descansado, asegurándose de que todos están hechos correctamente tras el cambio. Puede ayudar suponer que lo hemos hecho mal y buscar con el editor llamadas que aún estén sin cambiar.

Este es nuestro nuevo programa principal:

```
1     procedure main()
2          txt: TipoTxt;
3          cmd: TipoCmd;
4     {
5          txt = newtxt();
6          do{
7               saltarblancos(stdin);
8               if(eol()){
9                    readeol();
10              }else if(not eof()){
11                   readcmd(stdin, cmd);
12                   runcmd(cmd, txt);
13                   disposecmd(cmd);
14              }
15         }while(not eof());
16         disposetxt(txt);
17    }
```

Ignoramos las líneas en blanco y, si parece que tenemos un comando, lo leemos y llamamos a *runcmd* para ejecutarlo sobre el texto que tenemos. También llamamos a *disposecmd* para liberar la memoria de *cmd*, que tenía palabras dentro. Ahora podemos hacer un *runcmd* vacío, como este:

```
1     procedure runcmd(cmd: TipoCmd, ref txt: TipoTxt)
2     {
3           write(NombreCodigo[cmd.codigo]);
4           switch(cmd.codigo){
5           case Insertar, Borrar, Imprimir:
6                 write(' ');
7                 write(cmd.num[0]);
8                 write(' ');
9                 write(cmd.num[1]);
10          case Cambiar:
11                write(' ');
12                writepal(stdout, cmd.pal[0]);
13                write(' ');
14                writepal(stdout, cmd.pal[1]);
15          case Editar:
16                write(' ');
17                writepal(stdout, cmd.pal[0]);
18          }
19          writeeol();
20    }
```

Y empezar a probar comandos. A partir de ahora es cuestión de implementar un comando y, luego, reemplazar el código correspondiente en *runcmd* por la llamada al procedimiento que implementa el comando. Así uno por uno hasta tenerlos todos. Durante ese proceso es muy posible que tengamos que cambiar los argumentos de *runcmd*, en cuyo caso lo haremos así.

Verás también que hemos utilizado un *array* para guardar los nombres de los comandos: *NombreCodigo*. Aunque no lo mostramos aquí, también hemos cambiado *readcodigo* para que use ese *array* para leer el código de los comandos. Mira el listado final. La idea es que basta alterar el *array* para alterar el nombre de un comando en todo el programa. Usar los *arrays* de este modo, para que *definan* nombres que ve el usuario, es una idea muy popular.

Empezaremos por implementar el comando para editar un fichero y para imprimir líneas del mismo. Antes siquiera de empezar a implementarlo vemos que necesitamos algo que represente la edición de un fichero: no sólo el texto que tenemos, sino también el fichero correspondiente a dicho texto.

```
1           TipoEd = record
2           {
3                 nombrefich: TipoPal;
4                 txt: TipoTxt;
5           };
```

Dado que tenemos una edición en curso en todo momento (normalmente) y que podemos editar diversos ficheros a la vez, el programa principal tendrá que tener un conjunto de ediciones como variable local y pasarlo a *runcmd* para que lo utilice. Definimos un conjunto de ediciones:

```
1           TipoEds = array[0..MaxNumEd-1] of TipoEd;
2           TipoCjtoEds = record
3           {
4                 eds: TipoEds;
5                 numeds: int;
6                 actual: int;
7           };
```

Vamos a guardar en el *array eds* las ediciones que tenemos, y *numeds* indicará cuántas tenemos. Como sabemos ya que una de ellas será la edición en curso, el campo *actual* no servirá para indicar cuál. Tenemos que añadir una nueva constante *MaxNumEd* que limita el número máximo de ediciones y un procedimiento que inicializa nuestro conjunto:

```
1     procedure inicializarcjtoeds(ref ceds: TipoCjtoEds)
2     {
3          ceds.numeds = 0;
4          ceds.actual = Ninguno;
5     }
```

En el programa principal declaramos *ceds* como un conjunto de ediciones, llamamos a *inicializarcjtoeds* justo al principio, y pasamos *ceds* como argumento a *runcmd*. ¡Ah! Como una edición es algo que utiliza memoria dinámica (para el texto y el nombre de fichero) necesitamos también un *disposecjtoeds* y llamarlo al final del programa principal.

Ahora ya podemos implementar *editarcmd* para tener un nuevo comando para editar. Lo vamos a llamar desde *runcmd* tal y como se ve aquí:

```
1     procedure runcmd(cmd: TipoCmd, ref ceds: TipoCjtoEds)
2     {
3          switch(cmd.codigo){
           ...
13         case Editar:
14              editarcmd(cmd.pal[0], ceds);
           ...
21         }
22    }
```

Y este sería *editarcmd*:

```
1     procedure editarcmd(nomfich: TipoPal, ref ceds: TipoCjtoEds)
2          ed: TipoEd;
3     {
4          if(ceds.numeds == len ceds.eds){
5               writeln("no hay espacio para mas ediciones");
6          }else{
7               newed(ceds.eds[ceds.numeds], nomfich);
8               ceds.actual = ceds.numeds;
9               ceds.numeds = ceds.numeds + 1;
10              write("editando ");
11              writepal(stdout, nomfich);
12              writeeol();
13         }

15    }
```

El procedimiento se limita a elegir un lugar en *ceds.eds* para guardar una nueva edición, y actualizar tanto el índice para la edición actual como el número de ediciones. Avisamos también al usuario de que estamos editando un fichero. El trabajo lo hace realmente *newed*, que crea una nueva edición para el fichero cuyo nombre se le suministra como segundo argumento:

```
1     procedure newed(ref ed: TipoEd, pal: TipoPal)
2          txt: TipoTxt;
3          fich: file;
4          nomfich: TipoNomFich;
5     {
6          copiarpal(ed.nomfich, pal);
7          mknomfich(nomfich, pal);
8          open(fich, nomfich, "r");
9          readtxt(fich, ed.txt);
10         close(fich);
11    }
```

Dado que *open* necesita un *array* of *char*, utilizamos el procedimiento *mknomfich* y un *array* con tipo *TipoNomFich* para crear el nombre de fichero. Como el nombre de fichero puede tener cualquier tamaño, *mknomfich* ayuda a *open* para que sepa que sólo debe utilizar los primeros caracteres del array. Esto es poco limpio pero es parte del interfaz con el sistema operativo en Picky.

Para el comando para imprimir líneas implementamos un procedimiento *imprimircmd*:

```
1    procedure imprimircmd(rango: TipoRango, ref ceds: TipoCjtoEds)
2    {
3         if(ceds.actual == Ninguno){
4              writeln("no hay edicion en curso");
5         }else{
6              imprimirrango(ceds.eds[ceds.actual].txt, rango);
7         }
8    }
```

El trabajo lo hace *imprimirrango*.

```
1    procedure imprimirrango(txt: TipoTxt, rango: TipoArgNum)
2         lin: TipoLin;
3         i: int;
4    {
5         rangoporomision(txt, rango);
6         for(i = rango[0], i <= rango[1]){
7              writelin(stdout, getlin(txt, i));
8         }
9    }
```

Hemos necesitado acceder a la línea número ‘‘`i`’’ del texto. Para conseguirlo, hemos hecho algo similar a cuando necesitamos obtener el carácter n-ésimo de una palabra: inventarnos *getlin*. Otro problema es que el rango puede que utilizase números de línea que no existen en el texto. Para solucionar este problema, *rangoporomision* se ocupa de dar valores apropiados al comienzo y final del rango. Por ejemplo, si el rango tenía un sólo número, entonces el segundo se hace que sea igual al primero. Si cualquiera de ellos estaba fuera de rango en el texto, se cambia para que esto no sea así.

Esta es *getlin*:

```
1    function getlin(txt: TipoTxt, n: int): TipoLin
2         pnodo: TipoPtrTxt;
3         lin: TipoLin;
4    {
5         lin = newlin();
6         pnodo = txt.primero;
7         while(pnodo != nil and n >= 0){
8              lin = pnodo^.lin;
9              pnodo = pnodo^.siguiente;
10             n = n - 1;
11        }
12        return lin;
13   }
```

Nótese como siempre devuelve una línea correctamente formada, incluso si *n* está fuera de rango (devuelve una línea vacía en tal caso).

Para el comando de borrar líneas utilizaremos un nuevo procedimiento *borrarcmd*:

```
1     procedure borrarcmd(rango: TipoArgNum, ref ceds: TipoCjtoEds)
2     {
3          if(ceds.actual == Ninguno){
4               writeln("no hay edicion en curso");
5          }else{
6               borrarrango(ceds.eds[ceds.actual].txt, rango);
7          }
8     }
```

Este llamará a *borrarrango* de forma similar al comando anterior, y *borrarrango* llamará a *borrarlin* para cada número de línea en el rango. Este es *borrarlin*:

```
1     procedure borrarlin(ref txt: TipoTxt, i: int)
2          pnodo: TipoPtrTxt;
3          panterior: TipoPtrTxt;
4     {
5          pnodo = txt.primero;
6          panterior = nil;
7          while(pnodo != nil and i > 0){
8               panterior = pnodo;
9               pnodo = pnodo^.siguiente;
10              i = i - 1;
11         }

13         if(pnodo != nil){
14              if(panterior != nil){
15                   panterior^.siguiente = pnodo^.siguiente;
16              }else{
17                   txt.primero = pnodo^.siguiente;
18              }
19              if(txt.ultimo == pnodo){
20                   txt.ultimo = panterior;
21              }
22              disposelin(pnodo^.lin);
23              dispose(pnodo);
24         }
25    }
```

La primera parte del procedimiento se ocupa de localizar, en *pnodo*, el puntero al nodo correspondiente a la línea que hay que borrar. Además, dicha parte deja en *panterior* el puntero al nodo anterior (o *nil* si no lo hay). La segunda parte, sólo si *pnodo* no es nil (si la línea existía), ajusta los punteros del anterior (si lo hay) y los *primero* y *ultimo* del texto si es preciso. Además, libera la línea.

Imprimir el nombre de la edición actual es fácil. Para tomarnos un respiro, implementamos este comando ahora.

```
1     procedure cualcmd(ceds: TipoCjtoEds)
2     {
3          if(ceds.actual == Ninguno){
4               writeln("no hay edicion en curso");
5          }else{
6               writepal(stdout, ceds.eds[ceds.actual].nomfich);
7               writeeol();
8          }
9     }
```

El comando para listar todas las ediciones es similar, pero escribe todos los nombres de fichero en lugar del correspondiente a la edición actual.

El comando para guardar la edición en curso sólo ha de llamar a *writetxt* para guardar el texto. Aunque nosotros usamos algún otro procedimiento, ese es el efecto final. Tampoco lo reproducimos aquí. Mira el listado final.

El comando de insertar, *insertarcmd*, ha de borrar el rango identificado por los argumentos sólo cuando se han dado dos números. Después ha de insertar las líneas que se lean desde la entrada:

```
1     procedure insertarcmd(rango: TipoArgNum, ref ceds: TipoCjtoEds)
2           lin: TipoLin;
3     {
4           if(ceds.actual == Ninguno){
5                 writeln("no hay edicion en curso");
6           }else{
7                 if(rango[1] != Ninguno){
8                       borrarrango(ceds.eds[ceds.actual].txt, rango);
9                 }
10                insertarlineas(ceds.eds[ceds.actual].txt, rango[0]);
11          }
12    }
```

Para insertar las líneas tenemos que leer de *stdin*, una línea tras otra, e insertarlas conforme las leemos; parando de insertar sólo si la línea leida contiene sólo un ‘‘.’’ (también ante un fin de fichero, claro).

```
1     procedure insertarlineas(ref txt: TipoTxt, pos: int)
2           lin: TipoLin;
3           esfin: bool;
4     {
5           esfin = False;
6           while(not eof() and not esfin){
7                 readlin(stdin, lin);
8                 esfin = esfinlin(lin);
9                 if(esfin){
10                      disposelin(lin);
11                }else{
12                      inslin(txt, pos, lin);
13                }
14          }
15    }
```

Aquí, *esfinlin* se limita a comprobar que la línea tiene un sólo carácter ‘‘.’’. Astutamente, dado que la línea que leemos sólo la queremos para insertarla en el texto, suponemos que *inslin* se queda la línea, con lo que ni tenemos que copiarla ni hacer un *disposelin* si la insertamos en el texto.

La parte más complicada en realidad es *inslin*: Tendrá que hacer algo como lo que hacía *borrarlin*:

```
1           pnodo = txt.primero;
2           panterior = nil;
3           while(pnodo != nil and i > 0){
4                 panterior = pnodo;
5                 pnodo = pnodo^.siguiente;
6                 i = i - 1;
7           }
```

para situarse en la posición en la que hay que insertar la línea (y luego insertarla, claro). Como esto parece que lo necesitan tanto *borrarlin* como *inslin* y además es una operación que tiene sentido (corresponde a buscar una línea para insertar o borrar), hacemos un procedimiento nuevo en

lugar de repetir el código:

```
1     procedure buscarlin(ref txt: TipoTxt,
2                         i: int,
3                         ref pnodo: TipoPtrTxt,
4                         ref panterior: TipoPtrTxt)
5     {
6           pnodo = txt.primero;
7           panterior = nil;
8           while(pnodo != nil and i > 0){
9                 panterior = pnodo;
10                pnodo = pnodo^.siguiente;
11                i = i - 1;
12          }
13    }
```

Puede verse que, tras la llamada a *buscarlin*, puede que *pnodo* sea *nil* si la línea no está. Eso sí, *panterior* sólo será *nil* si la línea es la primera.

Así queda ahora *inslin*:

```
1     procedure inslin(ref txt: TipoTxt, i: int, lin: TipoLin)
2           pnodo: TipoPtrTxt;
3           panterior: TipoPtrTxt;
4           pnuevo: TipoPtrTxt;
5     {
6           buscarlin(txt, i, pnodo, panterior);
7           /* aunque pnodo sea nil, la insertamos siempre */
8           new(pnuevo);
9           pnuevo^.lin = lin;
10          pnuevo^.siguiente = nil;

12          if(panterior == nil){
13                pnuevo^.siguiente = txt.primero;
14                txt.primero = pnuevo;
15          }else{
16                pnuevo^.siguiente = panterior^.siguiente;
17                panterior^.siguiente = pnuevo;
18          }
19          if(txt.ultimo == nil or txt.ultimo == panterior){
20                txt.ultimo = pnuevo;
21          }
22    }
```

Nos falta por hacer el comando para reemplazar una palabra por otra, que es más fácil de lo que parece. Podemos recorrer todo el texto comparando cada una de las palabras con la que tenemos que cambiar. Cuando encontremos una palabra igual, la podemos cambiar por una copia de la palabra por la que hay que cambiarla. Pero antes de eso, hay que ver si podemos hacerlo:

```
1     procedure cambiarcmd(pals: TipoArgPal, ref ceds: TipoCjtoEds)
2     {
3           if(ceds.actual == Ninguno){
4                 writeln("no hay edicion en curso");
5           }else if(lenpal(pals[0]) == 0){
6                 writeln("no puedo reemplazar palabras vacias");
7           }else{
8                 txtcambiarpal(ceds.eds[ceds.actual].txt, pals[0], pals[1]);
9           }
10    }
```

Ahora, *txtcambiarpal* irá línea por línea cambiando palabras...

```
1     procedure txtcambiarpal(ref txt: TipoTxt, de: TipoPal, a: TipoPal)
2          pnodo: TipoPtrTxt;
3     {
4          pnodo = txt.primero;
5          while(pnodo != nil){
6               lincambiarpal(pnodo^.lin, de, a);
7               pnodo = pnodo^.siguiente;
8          }
9     }
```

Y *lincambiarpal* irá palabra por palabra cambiando palabras...

```
1     procedure lincambiarpal(ref lin: TipoLin, de: TipoPal, a: TipoPal)
2          pnodo: TipoPtrLin;
3     {
4          pnodo = lin.primero;
5          while(pnodo != nil){
6               if(igualpal(pnodo^.pal, de)){
7                    disposepal(pnodo^.pal);
8                    copiarpal(pnodo^.pal, a);
9               }
10              pnodo = pnodo^.siguiente;
11         }
12    }
```

12.10. Eso es todo

Y con esto tenemos nuestro flamante editor. Esta es una sesión de trabajo con el editor definitivo (aunque corresponde al código que mostramos tras el ejemplo de uso, que no es exactamente igual al que hemos contado hasta el momento tal y como te explicamos en el siguiente epígrafe).

```
; pick e.p
; out.pam
e etest
nueva edicion: etest
e etest2
nueva edicion: etest2

x
etest
etest2

f
etest2


e etest
editando etest


!p
1
2
3
4
```

```
i 2
hola
.

p
1
2
hola
3
4

w
etest guardado
```

El código del editor, después de probarlo un poco, ha quedado como sigue.

[e.p]

```
1   /*
2    * E, el editor definitivo.
3    */

5   program e;

7   consts:
8        Prueba = "una prueba";

10       Ninguno = -1;   /* para numeros de linea en TipoArgNum */
11       MaxNumEd = 16;  /* numero maximo de ficheros en edicion */
12       MaxNomFich = 64;     /* longitud maxima de nombres de fichero */
13       MaxNumArg = 2;  /* numero maximo de argumentos en un comando */

15       FinInsChar = '.';    /* caracter para fin de insercion */
16  types:
17       TipoQue = (Blancos, NoBlancos); /* argumento para readpal() */

19       /*
20        * Palabras (una lista de caracteres)
21        */
22       TipoPtrPal = ^TipoNodoPal;
23       TipoNodoPal = record
24       {
25            c: char;
26            siguiente: TipoPtrPal;
27       };
28       TipoPal = record
29       {
30            primero: TipoPtrPal;
31            ultimo: TipoPtrPal;
32       };
```

```
34          /*
35           * Lineas (una lista de palabras)
36           */
37          TipoPtrLin = ^TipoNodoLin;
38          TipoNodoLin = record
39          {
40               pal: TipoPal;
41               siguiente: TipoPtrLin;
42          };
43          TipoLin = record
44          {
45               primero: TipoPtrLin;
46               ultimo: TipoPtrLin;
47          };

49          /*
50           * Texto (una lista de lineas)
51           */
52          TipoPtrTxt = ^TipoNodoTxt;
53          TipoNodoTxt = record
54          {
55               lin: TipoLin;
56               siguiente: TipoPtrTxt;
57          };
58          TipoTxt = record
59          {
60               primero: TipoPtrTxt;
61               ultimo: TipoPtrTxt;
62               numlins: int;
63          };


66          /*
67           * Edicion
68           */
69          TipoEd = record
70          {
71               nomfich: TipoPal;
72               txt: TipoTxt;
73          };

75          TipoEds = array[0..MaxNumEd-1] of TipoEd;
76          TipoCjtoEds = record
77          {
78               eds: TipoEds;
79               numeds: int;
80               actual: int;
81          };

83          /* nombre de fichero a gusto del sistema operativo */
84          TipoNomFich = array[0..MaxNomFich-1] of char;

86          /*
87           * Comandos
88           */
89          TipoCodigo = (Insertar, Borrar, Imprimir, Cambiar, Editar,
90                        Guardar, Cual, Listar, Error);
91          TipoAridad = (NoArgs, Rango, Palabra, Palabras);
```

```
93          TipoArgNum = array[0..MaxNumArg-1] of int;
94          TipoArgPal = array[0..MaxNumArg-1] of TipoPal;

96          TipoCmd = record
97          {
98               codigo: TipoCodigo;
99               aridad: TipoAridad;
100              switch(aridad){
101              case Rango:
102                   num: TipoArgNum;          /* se usa solo si aridad es Rango. */
103              case Palabra, Palabras:
104                   pal: TipoArgPal;          /* solo si aridad es Palabra o Palabras. */
105              }
106         };

108         TipoNombreCodigo = array[TipoCodigo] of char;

110    consts:
111         /* nombres para cada comando, segun los ve el usuario */
112         NombreCodigo = TipoNombreCodigo('i', 'd', 'p', 's', 'e', 'w', 'f', 'x', '?');

114    function esblanco(c: char): bool
115    {
116         return c == ' ' or c == Tab;
117    }

119    function esdigito(c: char): bool
120    {
121         return c >= '0' and c <= '9';
122    }

124    function newpal(): TipoPal
125         pal: TipoPal;
126    {
127         pal.primero = nil;
128         pal.ultimo = nil;
129         return pal;
130    }

132    /* añadir un caracter a una palabra */
133    procedure appendcar(ref pal: TipoPal, c: char)
134         pnodo: TipoPtrPal;
135    {
136         new(pnodo);
137         pnodo^.c = c;
138         pnodo^.siguiente = nil;
139         if(pal.primero == nil){
140              pal.primero = pnodo;
141         }else{
142              pal.ultimo^.siguiente = pnodo;
143         }
144         pal.ultimo = pnodo;
145    }
```

```
147  function lenpal(pal: TipoPal): int
148       pnodo: TipoPtrPal;
149       n: int;
150  {
151       n = 0;
152       pnodo = pal.primero;
153       while(pnodo != nil){
154            n = n + 1;
155            pnodo = pnodo^.siguiente;
156       }
157       return n;
158  }

160  /*
161   * devolver el caracter n-esimo de pal, o Nul.
162   * Se empiezan a contar los caracteres en 0.
163   */
164  function getcar(pal: TipoPal, n: int): char
165       pnodo: TipoPtrPal;
166       c: char;
167  {
168       c = Nul;
169       pnodo = pal.primero;
170       while(pnodo != nil and n >= 0){
171            n = n - 1;
172            c = pnodo^.c;
173            pnodo = pnodo^.siguiente;
174       }
175       return c;
176  }

178  procedure writepal(ref fich: file, pal: TipoPal)
179       pnodo: TipoPtrPal;
180  {
181       pnodo = pal.primero;
182       while(pnodo != nil){
183            fwrite(fich, pnodo^.c);
184            pnodo = pnodo^.siguiente;
185       }
186  }

188  /* liberar la memoria dinamica que usa pal */
189  procedure disposepal(ref pal: TipoPal)
190       pnodo: TipoPtrPal;
191  {
192       while(pal.primero != nil){
193            pnodo = pal.primero;
194            pal.primero = pnodo^.siguiente;
195            dispose(pnodo);
196       }
197       pal.primero = nil;
198       pal.ultimo = nil;
199  }
```

```
procedure copiarpal(ref pal: TipoPal, orig: TipoPal)
     pnodo: TipoPtrPal;
{
     pal = newpal();
     pnodo = orig.primero;
     while(pnodo != nil){
          appendcar(pal, pnodo^.c);
          pnodo = pnodo^.siguiente;
     }
}

function igualpal(p1: TipoPal, p2: TipoPal): bool
     pnodo1: TipoPtrPal;
     pnodo2: TipoPtrPal;
     igual: bool;
{
     pnodo1 = p1.primero;
     pnodo2 = p2.primero;
     igual = True;
     while(igual and pnodo1 != nil and pnodo2 != nil){
          if(pnodo1^.c != pnodo2^.c){
               igual = False;
          }else{
               pnodo1 = pnodo1^.siguiente;
               pnodo2 = pnodo2^.siguiente;
          }
     }
     if((pnodo1 == nil and pnodo2 != nil) or
        (pnodo1 != nil and pnodo2 == nil)){
          return False;
     }else{
          return igual;
     }
}

/* convertir pal en un numero natural, val, o decir False en ok */
procedure convpalnat(ref pal: TipoPal, ref val: int, ref ok: bool)
     c: char;
     i: int;
{
     i = 0;
     val = 0;
     ok = True;
     while(ok and i < lenpal(pal)){
          c = getcar(pal, i);
          if(not esdigito(c)){
               ok = False;
          }else{
               val = val * 10;
               val = val + (int(c) - int('0'));
               i = i + 1;
          }
     }
}
```

```
function newlin(): TipoLin
     lin: TipoLin;
{
     lin.primero = nil;
     lin.ultimo = nil;
     return lin;
}

/* añadir una palabra a una linea */
procedure appendpal(ref lin: TipoLin, pal: TipoPal)
     pnodo: TipoPtrLin;
{
     new(pnodo);
     copiarpal(pnodo^.pal, pal);
     pnodo^.siguiente = nil;
     if(lin.primero == nil){
          lin.primero = pnodo;
     }else{
          lin.ultimo^.siguiente = pnodo;
     }
     lin.ultimo = pnodo;
}

procedure writelin(ref fich: file, lin: TipoLin)
     pnodo: TipoPtrLin;
{
     pnodo = lin.primero;
     while(pnodo != nil){
          writepal(fich, pnodo^.pal);
          pnodo = pnodo^.siguiente;
     }
     fwriteeol(fich);
}

/* liberar la memoria dinamica que utiliza una linea */
procedure disposelin(ref lin: TipoLin)
     pnodo: TipoPtrLin;
{
     while(lin.primero != nil){
          pnodo = lin.primero;
          lin.primero = pnodo^.siguiente;
          disposepal(pnodo^.pal);
          dispose(pnodo);
     }
     lin.primero = nil;
     lin.ultimo = nil;
}

procedure copiarlin(ref lin: TipoLin, orig: TipoLin)
     pnodo: TipoPtrLin;
{
     lin = newlin();
     pnodo = orig.primero;
     while(pnodo != nil){
          appendpal(lin, pnodo^.pal);
          pnodo = pnodo^.siguiente;
     }
}
```

```
function newtxt(): TipoTxt
     txt: TipoTxt;
{
     txt.primero = nil;
     txt.ultimo = nil;
     txt.numlins = 0;
     return txt;
}

/* añadir una linea al final de un texto */
procedure appendlin(ref txt: TipoTxt, lin: TipoLin)
     pnodo: TipoPtrTxt;
{
     new(pnodo);
     copiarlin(pnodo^.lin, lin);
     pnodo^.siguiente = nil;
     if(txt.primero == nil){
          txt.primero = pnodo;
          txt.ultimo = pnodo;
     }else{
          txt.ultimo^.siguiente = pnodo;
          txt.ultimo = pnodo;
     }
     txt.numlins = txt.numlins + 1;
}

/*
 * devolver la n-esima linea de pal, o una vacia.
 * Se empiezan a contar las lineas en 0.
 */
function getlin(txt: TipoTxt, n: int): TipoLin
     pnodo: TipoPtrTxt;
     lin: TipoLin;
{
     lin = newlin();
     pnodo = txt.primero;
     while(pnodo != nil and n >= 0){
          lin = pnodo^.lin;
          pnodo = pnodo^.siguiente;
          n = n - 1;
     }
     return lin;
}
```

```
/*
 * Una operacion auxiliar para implementar otras operaciones de texto.
 * Buscar la linea numero i, devolviendo un puntero al nodo de dicha
 * linea y uno al nodo anterior.
 * Si no existe el nodo (o el anterior) se devuelve nil.
 */
procedure buscarlin(ref txt: TipoTxt,
                i: int,
                ref pnodo: TipoPtrTxt,
                ref panterior: TipoPtrTxt)
{
     pnodo = txt.primero;
     panterior = nil;
     while(pnodo != nil and i > 0){
          panterior = pnodo;
          pnodo = pnodo^.siguiente;
          i = i - 1;
     }
}

/* eliminar la linea i-esima del texto (contando desde 0) */
procedure borrarlin(ref txt: TipoTxt, i: int)
     pnodo: TipoPtrTxt;
     panterior: TipoPtrTxt;
{
     buscarlin(txt, i, pnodo, panterior);

     if(pnodo != nil){
          if(panterior != nil){
               panterior^.siguiente = pnodo^.siguiente;
          }else{
               txt.primero = pnodo^.siguiente;
          }
          if(txt.ultimo == pnodo){
               txt.ultimo = panterior;
          }
          disposelin(pnodo^.lin);
          dispose(pnodo);
          txt.numlins = txt.numlins - 1;
     }
}

/* insertar directamente lin (no una copia) en la posicion i */
procedure inslin(ref txt: TipoTxt, i: int, lin: TipoLin)
     pnodo: TipoPtrTxt;
     panterior: TipoPtrTxt;
     pnuevo: TipoPtrTxt;
{
     buscarlin(txt, i, pnodo, panterior);
     /* aunque pnodo sea nil, la insertamos siempre */
     new(pnuevo);
     pnuevo^.lin = lin;
     pnuevo^.siguiente = nil;
```

```
413         if(panterior == nil){
414              pnuevo^.siguiente = txt.primero;
415              txt.primero = pnuevo;
416         }else{
417              pnuevo^.siguiente = panterior^.siguiente;
418              panterior^.siguiente = pnuevo;
419         }
420         if(txt.ultimo == nil or txt.ultimo == panterior){
421              txt.ultimo = pnuevo;
422         }
423         txt.numlins = txt.numlins + 1;
424    }

426    procedure writetxt(ref fich: file, txt: TipoTxt)
427         pnodo: TipoPtrTxt;
428    {
429         pnodo = txt.primero;
430         while(pnodo != nil){
431              writelin(fich, pnodo^.lin);
432              pnodo = pnodo^.siguiente;
433         }
434    }

436    /* liberar memoria dinamica que utiliza el texto */
437    procedure disposetxt(ref txt: TipoTxt)
438         pnodo: TipoPtrTxt;
439    {
440         while(txt.primero != nil){
441              pnodo = txt.primero;
442              txt.primero = pnodo^.siguiente;
443              disposelin(pnodo^.lin);
444              dispose(pnodo);
445         }
446         txt.primero = nil;
447         txt.ultimo = nil;
448         txt.numlins = 0;
449    }

451    /* leer una palabra formada por Blancos o NoBlancos, segun que */
452    procedure readpal(ref fich: file, ref pal: TipoPal, que: TipoQue)
453         c: char;
454         esnuestro: bool;
455    {
456         pal = newpal();
457         do{
458              fpeek(fich, c);
459              esnuestro = ((esblanco(c) and que == Blancos) or
460                   (not esblanco(c) and que == NoBlancos));
461              if(esnuestro){
462                   fread(fich, c);
463                   appendcar(pal, c);
464              }
465         }while(not feof(fich) and not feol(fich) and esnuestro);
466    }
```

```
468  procedure readlin(ref fich: file, ref lin: TipoLin)
469       pal: TipoPal;
470       c: char;
471  {
472       lin = newlin();
473       do{
474            fpeek(fich, c);
475            if(c != Eol and c != Eof){
476                 if(esblanco(c)){
477                      readpal(fich, pal, Blancos);
478                 }else{
479                      readpal(fich, pal, NoBlancos);
480                 }
481                 appendpal(lin, pal);
482                 disposepal(pal);
483            }
484       }while(not feof(fich) and not feol(fich));
485       if(feol(fich)){
486            freadeol(fich);
487       }
488  }

490  procedure readtxt(ref fich: file, ref txt: TipoTxt)
491       lin: TipoLin;
492       c: char;
493  {
494       txt = newtxt();
495       do{
496            readlin(fich, lin);
497            appendlin(txt, lin);
498            disposelin(lin);
499            fpeek(fich, c); /* hay eof? */
500       }while(not feof(fich));
501  }

503  /* leer el codigo de un comando, segun lo define NombreCodigo */
504  procedure readcodigo(ref fich: file, ref codigo: TipoCodigo)
505       c: char;
506       i: TipoCodigo;
507       esta: bool;
508  {
509       fread(fich, c);
510       codigo = Error;
511       i = Insertar;
512       while(i < Error and codigo == Error){
513            if(NombreCodigo[i] == c){
514                 codigo = i;
515            }else{
516                 i = succ(i);
517            }
518       }
519  }
```

```
/* Ojo! salta solo hasta el fin de linea, sin leerlo */
procedure saltarblancos(ref fich: file)
     c: char;
{
     do{
          fpeek(fich, c);
          if(esblanco(c)){
               fread(fich, c);
          }
     }while(not feof(fich) and not feol(fich) and esblanco(c));
}

function espalvacia(pal: TipoPal): bool
{
     return pal.primero == nil;
}

procedure inicializarcjtoeds(ref ceds: TipoCjtoEds)
{
     ceds.numeds = 0;
     ceds.actual = Ninguno;
}

/* liberar memoria dinamica utilizada por ed */
procedure disposeed(ref ed: TipoEd)
{
     disposetxt(ed.txt);
     disposepal(ed.nomfich);
}

/* liberar memoria dinamica utilizada por ceds */
procedure disposecjtoeds(ref ceds: TipoCjtoEds)
     i: int;
{
     for(i = 0, i < ceds.numeds){
          disposeed(ceds.eds[i]);
     }
}

/*
 * leer ninguno, uno o dos numeros.
 * Los que no se estan quedan con valor Ninguno.
 */
procedure readrangoargs(ref fich: file, ref arg: TipoArgNum, ref ok: bool)
     pal: TipoPal;
     i: int;
{
     arg[0] = Ninguno;
     arg[1] = Ninguno;
     ok = True;
```

```
572         saltarblancos(fich);
573         for(i = 0, i < len arg){
574              saltarblancos(fich);
575              if(not feof(fich) and not feol(fich)){
576                   readpal(fich, pal, NoBlancos);
577                   convpalnat(pal, arg[i], ok);
578                   disposepal(pal);
579              }
580         }
581    }

583    /* Leer dos palabras de la forma XunaXotraX, donde X es un char */
584    procedure readpalsarg(ref fich: file, ref pals: TipoArgPal, ref ok: bool)
585         separador: char;
586         c: char;
587         i: int;
588    {
589         separador = '/';          /* no se usa este valor en realidad */

591         for(i = 0, i < len pals){
592              ok = not feof(fich) and not feol(fich);
593              if(ok){
594                   fread(fich, separador);
595                   ok = not feof(fich) and not feol(fich);
596              }

598              pals[i] = newpal();

600              do{
601                   fpeek(fich, c);
602                   ok = not feof(fich) and not feol(fich);
603                   if(ok and c != separador){
604                        fread(fich, c);
605                        appendcar(pals[i], c);
606                   }
607              }while(ok and c != separador);
608         }

610         if(ok){
611              fread(fich, separador);
612         }
613    }

615    /* leer todo lo que quede en la linea, incluyendo Eol */
616    procedure saltarlinea(ref fich: file)
617         c: char;
618    {
619         while(not feof(fich) and not feol(fich)){
620              fread(fich, c);
621         }
622         if(feol(fich)){
623              freadeol(fich);
624         }
625    }
```

```
627  procedure readpalarg(ref fich: file, ref pal: TipoPal, ref ok: bool)
628  {
629       ok = False;
630       saltarblancos(fich);
631       ok = not feof(fich) and not feol(fich);
632       if(ok){
633            readpal(fich, pal, NoBlancos);
634       }
635  }

637  procedure readcmd(ref fich: file, ref cmd: TipoCmd)
638       ok: bool;
639  {
640       readcodigo(fich, cmd.codigo);
641       if(cmd.codigo != Error){
642            ok = True;
643            switch(cmd.codigo){
644            case Insertar, Borrar, Imprimir:
645                 readrangoargs(fich, cmd.num, ok);
646            case Cambiar:
647                 readpalsarg(fich, cmd.pal, ok);
648            case Editar:
649                 readpalarg(fich, cmd.pal[0], ok);
650            }
651            if(not ok){
652                 cmd.codigo = Error;
653            }
654       }
655       saltarlinea(fich);
656  }

658  /* Truculento! crear un nombre de fichero para el S.O. */
659  procedure mknomfich(ref nom: TipoNomFich, pal: TipoPal)
660       pnodo: TipoPtrPal;
661       i: int;
662  {
663       if(lenpal(pal) > len nom){
664            fatal("TipoNomFich ha de ser mas grande");
665       }
666       for(i = 0, i < lenpal(pal)){
667            nom[i] = getcar(pal, i);
668       }
669       /*
670        * puede que lenpal(pal) sea menor que len nom
671        * si es asi, hay que hacer que open() sepa que solo
672        * debe usar los primeros caracteres del array que le pasamos;
673        * en picky eso se hace poniendo Nul al final de estos caracteres.
674        * El interfaz con el Sistema Operativo es asi...
675        */
676       nom[lenpal(pal)] = Nul;
677  }
```

```
679  procedure newed(ref ed: TipoEd, pal: TipoPal)
680       txt: TipoTxt;
681       fich: file;
682       nomfich: TipoNomFich;
683  {
684       copiarpal(ed.nomfich, pal);
685       mknomfich(nomfich, pal);
686       open(fich, nomfich, "r");
687       readtxt(fich, ed.txt);
688       close(fich);
689  }

691  procedure guardared(ref ed: TipoEd)
692       txt: TipoTxt;
693       fich: file;
694       nomfich: TipoNomFich;
695  {
696       mknomfich(nomfich, ed.nomfich);
697       open(fich, nomfich, "w");
698       writetxt(fich, ed.txt);
699       close(fich);
700       writepal(stdout, ed.nomfich);
701       writeln(" guardado");
702  }

704  procedure editar(ref ceds: TipoCjtoEds, nomfich: TipoPal, ref ok: bool)
705       i: int;
706  {
707       ok = False;
708       i = 0;
709       while(i < ceds.numeds and not ok){
710            if(igualpal(ceds.eds[i].nomfich, nomfich)){
711                 ceds.actual = i;
712                 ok = True;
713            }else{
714                 i = i + 1;
715            }
716       }
717  }

719  procedure editarcmd(nomfich: TipoPal, ref ceds: TipoCjtoEds)
720       ed: TipoEd;
721       ok: bool;
722  {
723       if(ceds.numeds == len ceds.eds){
724            writeln("no hay espacio para mas ediciones");
725       }else{
726            editar(ceds, nomfich, ok);
727            if(ok){
728                 write("editando ");
729                 writepal(stdout, nomfich);
730                 writeeol();
731            }else{
```

```
733                 newed(ceds.eds[ceds.numeds], nomfich);
734                 ceds.actual = ceds.numeds;
735                 ceds.numeds = ceds.numeds + 1;
736                 write("nueva edicion: ");
737                 writepal(stdout, nomfich);
738                 writeeol();
739           }
740      }

742  }

744  procedure guardarcmd(ref ceds: TipoCjtoEds)
745  {
746       if(ceds.actual == Ninguno){
747            writeln("no hay edicion en curso");
748       }else{
749            guardared(ceds.eds[ceds.actual]);
750       }
751  }

753  function lentxt(txt: TipoTxt): int
754  {
755       return txt.numlins;
756  }

758  /*
759   * Dejar los numeros de linea en rango con valores razonables para
760   * el texto. Incluso si alguno de dichos numeros era Ninguno.
761   */
762  procedure rangoporomision(txt: TipoTxt, ref rango: TipoArgNum)
763  {
764       if(rango[0] == Ninguno and rango[1] == Ninguno){
765            rango[0] = 0;
766            rango[1] = lentxt(txt) - 1;
767       }
768       if(rango[0] < 0){
769            rango[0] = 0;
770       }else if(rango[0] > lentxt(txt) - 1){
771            rango[0] = lentxt(txt) - 1;
772       }
773       if(rango[1] == Ninguno){
774            rango[1] = rango[0];
775       }else if(rango[1] > lentxt(txt) - 1){
776            rango[1] = lentxt(txt) - 1;
777       }
778  }

780  procedure imprimirrango(txt: TipoTxt, rango: TipoArgNum)
781       lin: TipoLin;
782       i: int;
783  {
784       rangoporomision(txt, rango);
785       for(i = rango[0], i <= rango[1]){
786            writelin(stdout, getlin(txt, i));
787       }
788  }
```

```
790  procedure borrarrango(ref txt: TipoTxt, rango: TipoArgNum)
791       plin: TipoPtrTxt;
792       i: int;
793  {
794       rangoporomision(txt, rango);
795       for(i = rango[0], i <= rango[1]){
796            borrarlin(txt, rango[0]);
797       }
798  }

800  function esfinpal(pal: TipoPal): bool
801  {
802       if(pal.primero != pal.ultimo){
803            return False;
804       }else if(pal.primero == nil){
805            return True;
806       }else{
807            return pal.primero^.c == FinInsChar;
808       }
809  }

811  function esfinlin(lin: TipoLin): bool
812  {
813       if(lin.primero != lin.ultimo){
814            return False;
815       }else if(lin.primero == nil){
816            return True;
817       }else{
818            return esfinpal(lin.primero^.pal);
819       }
820  }

822  /* leer lineas hasta una que es "." e insertarlas en pos en texto */
823  procedure insertarlineas(ref txt: TipoTxt, pos: int)
824       lin: TipoLin;
825       esfin: bool;
826  {
827       esfin = False;
828       while(not eof() and not esfin){
829            readlin(stdin, lin);
830            esfin = esfinlin(lin);
831            if(esfin){
832                 disposelin(lin);
833            }else{
834                 inslin(txt, pos, lin);
835                 pos = pos + 1;  /* la siguiente detras */
836            }
837       }
838  }

840  procedure imprimircmd(rango: TipoArgNum, ref ceds: TipoCjtoEds)
841  {
842       if(ceds.actual == Ninguno){
843            writeln("no hay edicion en curso");
844       }else{
845            imprimirrango(ceds.eds[ceds.actual].txt, rango);
846       }
847  }
```

```
849  procedure borrarcmd(rango: TipoArgNum, ref ceds: TipoCjtoEds)
850  {
851       if(ceds.actual == Ninguno){
852            writeln("no hay edicion en curso");
853       }else{
854            borrarrango(ceds.eds[ceds.actual].txt, rango);
855       }
856  }

858  procedure cualcmd(ceds: TipoCjtoEds)
859  {
860       if(ceds.actual == Ninguno){
861            writeln("no hay edicion en curso");
862       }else{
863            writepal(stdout, ceds.eds[ceds.actual].nomfich);
864            writeeol();
865       }
866  }

868  procedure listarcmd(ceds: TipoCjtoEds)
869       i: int;
870  {
871       if(ceds.numeds == 0){
872            writeln("no hay ediciones");
873       }else{
874            for(i = 0, i < ceds.numeds){
875                 writepal(stdout, ceds.eds[i].nomfich);
876                 writeeol();
877            }
878       }
879  }

881  procedure insertarcmd(rango: TipoArgNum, ref ceds: TipoCjtoEds)
882       lin: TipoLin;
883  {
884       if(ceds.actual == Ninguno){
885            writeln("no hay edicion en curso");
886       }else{
887            if(rango[1] != Ninguno){
888                 borrarrango(ceds.eds[ceds.actual].txt, rango);
889            }
890            insertarlineas(ceds.eds[ceds.actual].txt, rango[0]);
891       }
892  }

894  procedure lincambiarpal(ref lin: TipoLin, de: TipoPal, a: TipoPal)
895       pnodo: TipoPtrLin;
896  {
897       pnodo = lin.primero;
898       while(pnodo != nil){
899            if(igualpal(pnodo^.pal, de)){
900                 disposepal(pnodo^.pal);
901                 copiarpal(pnodo^.pal, a);
902            }
903            pnodo = pnodo^.siguiente;
904       }
905  }
```

```
907   procedure txtcambiarpal(ref txt: TipoTxt, de: TipoPal, a: TipoPal)
908         pnodo: TipoPtrTxt;
909   {
910         pnodo = txt.primero;
911         while(pnodo != nil){
912               lincambiarpal(pnodo^.lin, de, a);
913               pnodo = pnodo^.siguiente;
914         }
915   }

917   procedure cambiarcmd(pals: TipoArgPal, ref ceds: TipoCjtoEds)
918   {
919         if(ceds.actual == Ninguno){
920               writeln("no hay edicion en curso");
921         }else if(lenpal(pals[0]) == 0){
922               writeln("no puedo reemplazar palabras vacias");
923         }else{
924               txtcambiarpal(ceds.eds[ceds.actual].txt, pals[0], pals[1]);
925         }
926   }

928   procedure runcmd(cmd: TipoCmd, ref ceds: TipoCjtoEds)
929   {
930         switch(cmd.codigo){
931         case Insertar:
932               insertarcmd(cmd.num, ceds);
933         case Borrar:
934               borrarcmd(cmd.num, ceds);
935         case Imprimir:
936               imprimircmd(cmd.num, ceds);
937         case Cambiar:
938               cambiarcmd(cmd.pal, ceds);
939         case Editar:
940               editarcmd(cmd.pal[0], ceds);
941         case Guardar:
942               guardarcmd(ceds);
943         case Cual:
944               cualcmd(ceds);
945         case Listar:
946               listarcmd(ceds);
947         }
948   }

950   /* liberar la memoria dinamica que utiliza cmd */
951   procedure disposecmd(ref cmd: TipoCmd)
952   {
953         switch(cmd.codigo){
954         case Cambiar:
955               disposepal(cmd.pal[0]);
956               disposepal(cmd.pal[1]);
957         case Editar, Guardar:
958               disposepal(cmd.pal[0]);
959         }
960   }
```

```
962   /*
963    * Esto es e!
964    */
965   procedure main()
966        ceds: TipoCjtoEds;
967        cmd: TipoCmd;
968   {
969        inicializarcjtoeds(ceds);
970        do{
971             saltarblancos(stdin);
972             if(eol()){
973                  readeol();
974             }else if(not eof()){
975                  readcmd(stdin, cmd);
976                  runcmd(cmd, ceds);
977                  disposecmd(cmd);
978             }
979        }while(not eof());
980        disposecjtoeds(ceds);
981   }
—
```

12.11. ¿Y ya está?

Desde luego que no. Lo primero sería probar *de verdad* el editor. Seguramente queden errores o, como suele llamárseles, *bugs* dentro del código. Nosotros lo hemos hecho y hemos descubierto algunos errores que nos habían pasado inadvertidos al escribir el programa y, por tanto, no hemos contado anteriormente. No obstante, los que hemos detectado están arreglados en el código. El más notable es que *editarcmd* no se daba cuenta de si ya estaba editando un fichero o no: siempre cargaba un nuevo fichero para editar.

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.

Aunque no esté roto, es seguro que los comandos que acepta el editor no son *exactamente* los que habíamos especificado. Hay muchas decisiones de implementación (como el uso de rangos) que pueden alterar el interfaz de los comandos. Es posible que el editor admita algún comando en el que no habíamos pensado (por ejemplo, prueba a insertar sin indicar ningún número de línea), y es posible que el comportamiento no sea exactamente el que habíamos imaginado antes de programarlo. En cualquier caso, deberíamos utilizarlo un tiempo y actualizar su manual y/o su especificación para que corresponda a la implementación.

Por otra parte hay muchas cosas que el editor está haciendo de un modo muy ineficaz. Por ejemplo, muchas palabras se están creando sólo para pasarlas a unos cuantos procedimientos, que las copian y las insertan, y luego la palabras originales se destruyen. ¿No sería mucho mejor hacer que las que creamos sean directamente las que se utilizan? Nos ahorramos crear y destruir otras.

O por ejemplo, hay muchos sitios en el código donde se llama a la misma función con el mismo argumento múltiples veces en el mismo subprobrama. Es mucho más eficaz utilizar una variable local y llamar a la función una sóla vez.

Hay muchos otros ejemplos... 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 la mayor parte del 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.

Índice Analítico

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.