Introducción a la Programación usando Pascal como primer lenguaje

Capítulo 9
Lectura de Ficheros

Francisco J. Ballesteros

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 Pascal los ficheros están representados por el tipo de datos **`file of ...`**, donde el tipo que sigue a "**`file of`**" indica qué registros contiene el fichero. Lo normal es utilizar **`byte`** como tipo elemento la mayoría de las veces.

No obstante, un caso muy habitual es guardar caracteres empleando dichos bytes. El tipo de datos "**`text`**" representa un fichero de caracteres que se utiliza para guardar texto. Pero recuerda que todos los fichero guardan simplemente bytes.

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.
- Después podemos leer o escribir en el fichero.
- Cuando terminemos hemos de cerrar el fichero.

En nuestro dialecto de Pascal, abrir un fichero consiste en llamar a dos

procemientos:

- **assign** asigna un nombre a la variable de tipo fichero para que Pascal sepa qué fichero corresponde a la variable.
- **reset** prepara el fichero para leerlo desde el principio y **rewrite** lo prepara para (re)escribirlo

Cerrarlo es más normal y consiste en llamar al procedimiento **close** para terminar de trabajar con el fichero.

El siguiente programa abre un fichero llamado **texto**, lee una línea de dicho fichero (50 caracteres máximo) y la escribe en la salida tras cerrar el fichero.

```
 1
 2 {
 3     Leer un saludo del fichero
 4
 5 }
 6
 7 program leerlinea;
 8
 9 const
10     MaxStr = 50;
11
12 type
13     TipoStr = string[MaxStr];
14
15 var
16     entrada: text;    { basicamente file of char }
17     linea: TipoStr;
18 begin
19     assign(entrada, 'texto');
20     reset(entrada);
21     read(entrada, linea);
22     close(entrada);
23     writeln(output, linea);
24 end.
```

El primer argumento de **assign**, es el fichero tal y como lo representa Pascal, esto es, la variable **entrada**. El segundo argumento es el nombre del fichero tal y como lo conoce el sistema operativo. La línea 20 deja el fichero preparado para su lectura desde el principio. A partir de esta sentencia podemos utilizar la variable de tipo **text** para referirnos al fichero llamado **texto** 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. Pero ten en cuenta que el nombre habitualmente

ha de representar el camino hacia el fichero en el disco, por ejemplo, **`/home/nemo/texto`**, o puede corresponder a distintos ficheros si no se indica un camino, como en el programa del ejemplo en que usamos **`texto`** y ejecutamos el programa en un directorio (o carpeta) que tiene un fichero con dicho nombre.

El procedimiento **`read`** acepta como primer argumento el fichero desde el que queremos leer, utilizando la entrada estándar si sólo se le suministra el segundo argumento (la variable que queremos que contenga los datos que leemos).

Como nota curiosa, la entrada estándar ya está abierta desde el comienzo de la ejecución del programa sin que sea preciso invocar a ningún procedimiento. Igual sucede con la salida estándar. Pascal incluye dos variables predefinidas, **`input`** y **`output`**, ambas de tipo *file*, que corresponden a la entrada y a la salida del programa.

Si miras la línea 23 del programa, notarás que a **`writeln`** le hemos suministrado como primer argumento la variable global **`output`**, lo que hace que escriba en dicho fichero. Hacer esto es equivalente a no suministrar el fichero como primer argumento, dado que en dicho caso **`writeln`** escribe en la salida estándar, esto es, en **`output`**.

La práctica totalidad de los procedimientos y funciones que leen, escriben e informan sobre ficheros operan del mismo modo. Aceptan como primer argumento un fichero y trabajan con la entrada o la salida estándar si dicho argumento no se suministra.

Para guardar datos en un fichero tendremos que abrirlo para escribir y no para leer. Esto se hace utilizando **`rewrite`** en lugar de **`reset`**. En otros lenguajes, un único procedimiento **`open`** se ocupa de abrir el fichero para leer, para escribir o para ambas cosas.

Una vez hemos terminado de utilizar un fichero es preciso llamar al procedimiento **`close`** para **cerrar** el fichero. 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.

Recuerda una vez mas que aunque puedes utilizar **`read`**, **`write`**, **`readln`** y **`writeln`** para leer de la entrada y escribir en la salida estándar, si suministras como primer argumento un fichero, trabajarán con dicho fichero.

Por ejemplo, este programa abre un fichero llamado **`datos.txt`** y reemplaza su contenido por una única línea que contiene el texto "**`hola`**"

```
1
2 {
3     Escribir un saludo en un fichero
```

```
 4
 5 }
 6
 7 program escrlinea;
 8
 9 var
10     salida: text;    { basicamente file of char }
11 begin
12     assign(salida, 'datos.txt');
13     rewrite(salida);
14     writeln(salida, 'hola');
15     close(salida);
16 end.
```

Por convenio, siempre se escribe un fin de línea tras cada línea de texto. Igualmente, se supone que siempre existe un fin de línea tras cada línea de texto en cualquier fichero de texto que utilicemos como datos de entrada. También podríamos haber hecho esto:

```
 1
 2 {
 3     Escribir un saludo en un fichero
 4
 5 }
 6
 7 program escrlinea;
 8
 9 var
10     salida: text;    { basicamente file of char }
11 begin
12     assign(salida, 'datos.txt');
13     rewrite(salida);
14     write(salida, 'hola');
15     writeln(salida);
16     close(salida);
17 end.
```

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. El fichero que utilizamos como entrada tiene este texto:

```
A long time ago.. before Disney came...
Star Wars was great
```

El texto de cada línea comienza justo al principio de la línea. El programa es como

sigue:

```
 1
 2 {
 3     Leer un saludo del fichero
 4 }
 5
 6 program leerlinea;
 7
 8 const
 9     MaxStr = 4;
10
11 type
12     TipoStr = string[MaxStr];
13
14 var
15     entrada: text;    { basicamente file of char }
16     linea: TipoStr;
17 begin
18     assign(entrada, 'texto');
19     reset(entrada);
20     read(entrada, linea);
21     close(entrada);
22     writeln('[', linea, ']');
23 end.
```

Cuando ejecutamos nuestro programa podemos ver lo que pasa:

```
unix% leerlinea
[A lo]
```

Como podrás ver, Pascal ha leído los primeros cuatro caracteres y los ha dejado en nuestra variable **linea**, sin leer nada más aunque exista en el fichero. Vamos a modificar un poco el programa para que quede como este

```
begin
    assign(entrada, 'texto');
    reset(entrada);
    read(entrada, linea);
    writeln('[', linea, ']');
    read(entrada, linea);
    writeln('[', linea, ']');
    close(entrada);
end.
```

Y ahora, al ejecutarlo vemos esto:

```
unix% leerlinea
[A lo]
[ng t]
```

Como podrás ver, la segunda lectura ha leido los siguientes cuatro caracteres del fichero.

Si ahora hacemos lo mismo dejando en el fichero una sóla "**A**'' entonces vemos esta salida al ejecutar:

```
unix% leerlinea
[A]
[]
```

y comprobamos que Pascal no ha podido leer nada más. La lectura de strings funciona así. Al leer arrays de caracteres, el funcionamiento es igual. Pascal salta los fines de línea y lee tantos caracteres como pueda. Si puede leer menos de los que queremos, entonces, ¡Mala suerte!

La cosa cambia si intentamos leer un entero, por ejemplo. Vamos a utilizar ahora el siguiente programa:

```
 1
 2 {
 3     Leer un numero del fichero
 4 }
 5
 6 program leerint;
 7
 8 const
 9     MaxStr = 4;
10
11 type
12     TipoStr = array [1..4] of char;
13
14 var
15     entrada: text;    { basicamente file of char }
16     n: integer;
17 begin
18     assign(entrada, 'datos');
19     reset(entrada);
20     read(entrada, n);
21     writeln(n);
22     close(entrada);
23 end.
```

Y como entrada, vamos a crear un fichero llamado "**datos**'' que tenga una línea con algo de espacio en blanco y, en la segunda línea, el número "**666**'' escrito con texto. Si ejecutamos el programa vemos que todo está bien:

```
unix% leerint
666
```

Pascal ha saltado el espacio en blanco hasta encontrar el número, y lo ha leído. Pero probemos ahora si borramos el número y dejamos sólo el espacio en blanco en el fichero:

```
unix% leerint
0
```

No hemos podido leer el número. En muchos lenguajes, incluyendo muchos dialectos de Pascal, habríamos sufrido un error en tiempo de ejecución si al intentar leer algo no lo tenemos disponible. Este dialecto es algo más permisivo, con lo que hemos de tener más cuidado si queremos estar seguros de lo que leemos. De otro modo, ¿Cómo podríamos saber si el fichero tenía un "**0**'' escrito o es que estaba vacío?

Aprenderemos más sobre lectura controlada en el resto de este tema.

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 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.

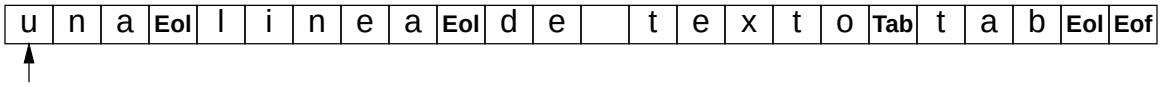

Fig. 1. Fichero del que lee el programa.

Podemos ver que al final de cada línea hay una marca de fin de línea representada por **Eol** (de *end of line*) en la figura. 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 entonces estaremos en la situación que ilustra la figura 2.

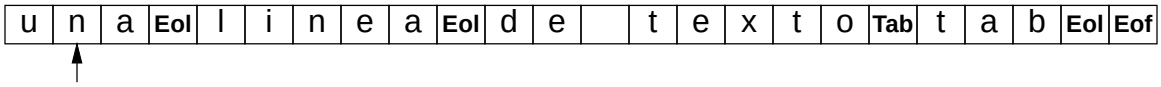

Fig. 2. Fichero tras leer un carácter.

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 3.

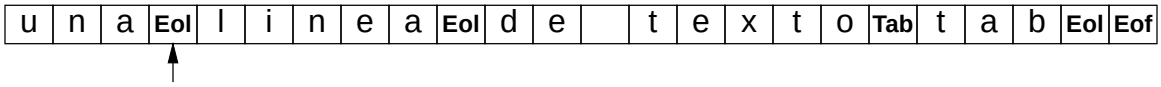

Fig. 3. Fichero tras haber leído dos caracteres más.

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 "**`l`**'' como podríamos creer. En lugar de eso, leemos basura (algo que no esperamos, todo o parte del fin de línea). Podemos comprobar esto si hacemos un programa que lea varios caracteres y nos escriba en cada ocasión el carácter leído y el valor devuelto una función llamada "**`eoln`**'' que, en Pascal, devuelve "**`true`**'' si se ha encontrado el fin de

línea.

```
 1
 2 {
 3     Jugar con EOLN
 4 }
 5
 6 program leercars;
 7
 8 var
 9     entrada: text;    { basicamente file of char }
10     c: char;
11 begin
12     assign(entrada, 'datos');
13     reset(entrada);
14     read(entrada, c);
15     writeln('[', c, '] ', eoln(entrada));
16     read(entrada, c);
17     writeln('[', c, '] ', eoln(entrada));
18     read(entrada, c);
19     writeln('[', c, '] ', eoln(entrada));
20     read(entrada, c);
21     writeln('[', c, '] ', eoln(entrada));
22     close(entrada);
23 end.
```

Si lo ejecutamos con un fichero que tiene este contenido:

```
un
lin
```

entonces esto es lo que vemos:

```
unix% leercars
[u] FALSE
[n] TRUE
[
] FALSE
[l] FALSE
```

Curiosamente, escribir el carácter que hemos leído al estar en el fin de línea, ha provocado un salto de línea en la salida. Esto es así en UNIX, pero en otros sistemas la cosa puede ser diferente. En Windows por ejemplo, el fin de línea tiene dos caracteres.

Lo normal al llegar a un fin de línea, es saltarlo. Para eso podemos utilizar "**readln**'' como ya sabemos. Por ejemplo, este otro programa

```
 1
 2 {
 3     Jugar con EOLN
 4 }
 5
 6 program leercars;
 7
 8 var
 9     entrada: text;    { basicamente file of char }
10     c: char;
11 begin
12     assign(entrada, 'datos');
13     reset(entrada);
14     read(entrada, c);
15     writeln('[', c, '] ', eoln(entrada));
16     read(entrada, c);
17     writeln('[', c, '] ', eoln(entrada));
18     readln(entrada);
19     read(entrada, c);
20     writeln('[', c, '] ', eoln(entrada));
21     read(entrada, c);
22     writeln('[', c, '] ', eoln(entrada));
23     close(entrada);
24 end.
```

produce esta salida con la misma entrada:

```
unix% leercars
[u] FALSE
[n] TRUE
[l] FALSE
[i] FALSE
```

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 **`readln`**.

Siguiendo con nuestro ejemplo, al llamar a **`readln`** la posición del fichero salta el fin de línea y pasa a estar en el siguiente carácter.

¿Cómo sabemos si tenemos que saltar un fin de línea? Es fácil. Como has visto, "**`eoln`**'' devuelve "**`true`**'' si estamos mirando al fin de línea en el fichero. Podemos utilizarla con un fichero, o, si no pasamos argumentos, con la entrada estándar.

Por cierto, para escribir un fin de línea, necesitamos usar el procedimiento **`writeln`** como hemos hecho muchas veces.

Seguimos con nuestro anterior ejemplo. Una vez se ha saltado el fin de línea, nuestro fichero quedaría como se ven la figura:

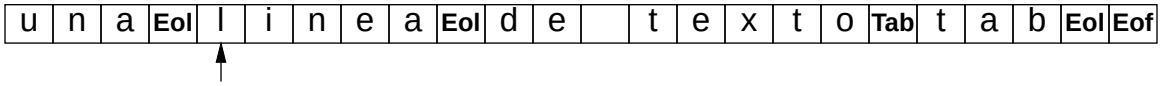

Fig. 4. Tras EOL

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 o deba leer. Esta vez, nuestro dialecto de Pascal normalmente deja de leer si no puede leer nada más. La función predefinida **`eof`** devuelve "**`true`**'' cuando estamos ante el fin de fichero. En ese caso, tenemos que dejar de leer el fichero. En otros lenguajes y dialectos, intentar leer llegado al fin de fichero supone un error en tiempo de ejecución.

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 UNIX, el sistema operativo hace que la lectura de teclado se encuentre un fin de fichero, siempre que no tengamos nada escrito en la línea.

Otra nota de atención. Si llamamos a **`eof`** o a **`eoln`** con la entrada estándar, veremos que el program se para hasta que escribimos algo en el teclado y pulsamos *Enter*. Pascal está intentando leer dado que no es adivino y no puede saber si hay un fin de línea o fichero antes de intentar leerlo.

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.

```
1
2 {
3     Copiar un fichero en la salida
```

```
 4
 5 }
 6
 7 program copia;
 8
 9 procedure copiar(var entrada, salida: Text);
10 var
11     c: char;
12 begin
13     while not eof(entrada) do begin
14         if eoln(entrada) then begin
15             readln(entrada);
16             writeln(salida);
17         end
18         else begin
19             read(entrada, c);
20             write(salida, c);
21         end;
22     end;
23 end;
24
25 var
26     entrada: Text;
27 begin
28     assign(entrada, 'texto');
29     reset(entrada);
30     copiar(entrada, output);
31     close(entrada);
32 end.
```

Un detalle importante es que el programa no intenta siquiera leer los fines de línea presentes en el fichero. Así, si **`eoln`** indica que estamos mirando al fin de línea, el programa evita dicha marca de fin de línea llamando a **`readln`** y después escribe el fin de línea llamando a **`writeln`**.

Observa que **los ficheros se pasan siempre por referencia**. Pasarlos por valor suele ser un desastre. Son entidades externas y Pascal no tiene el control sobre ellos.

9.3. Lectura controlada

Cuando leemos una cadena de caracteres, **`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 leerá el fin de línea (algo inesperado) y quizá parte de la siguiente línea. Desde luego no queremos que eso ocurra.

Cuando llamamos a **read** 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). Naturalmente, para saltarse los blancos, los lee.

Sucede lo mismo si intentamos leer un valor de un tipo enumerado. **read** salta espacios en blanco y luego intenta leer caracteres que formen el nombre del literal del tipo enumerado.

Si al leer un número o un enumerado no tenemos en la entrada algo que corresponda a un número o a un enumerado, el programa sufrirá un error en tiempo de ejecución en la mayoría de los casos.

¿Qué podemos hacer entonces si queremos leer controladamente un fichero? La solución es leer carácter a carácter teniendo cuidado de comprobar si lo que hemos leído es lo que esperamos, y prestando atención a los fines de línea para saltarlos o parar según deseemos. Todo ello, sin pasar el fin de fichero.

Queremos leer controladamente la entrada para escribir a la salida una palabra por línea. Suponemos que las palabras están formadas por cualquier cosa que no sea un blanco.

El problema se puede hacer leyendo carácter a carácter. 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, *TipoStr*, y cuantos subprogramas podamos necesitar para manipularlo. Estando así las cosas podríamos escribir:

```
var
    str: TipoStr;
begin
    while not eof() do begin
        leerstr(input, str);
        if longstr(str) > 0 then begin
            escrstrln(str);
        end;
    end;
end.
```

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 leer una sóla palabra y escribirla. El programa sería

saltarnos los caracteres que forman una palabra en lugar de leer una palabra. Nuestro programa podría quedar así:

```
var
    str: TipoStr;
begin
    leerstr(input, str);
    escrstrln(str);
end.
```

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 declararnos nuestro tipo de datos:

Vamos a utilizar el mismo esquema que hemos utilizado antes para copiar la entrada en la salida.

```
const
    LongStr = 50;
type
    TipoStr = record
        cars: string[LongStr];
        ncars: integer;
    end;
```

Guardamos un máximo de 50 caracteres y recordamos cuántos hemos guardado. Un tipo de datos similar se utiliza siempre que tenemos colecciones de tamaño variable.

Para leer una palabra, vaciamos la palabra, leemos todo el espacio en blanco que encontremos y luego los caracteres de la palabra, que guardamos en nuestra variable.

```
procedure leerstr(var entrada: Text; var str: TipoStr);
```

```
    var
        c: char;
        haypalabra: boolean;
    begin
        vaciarstr(str);
        haypalabra := false;
        while not eof(entrada) and not haypalabra do begin
            if eoln(entrada) then begin
                readln(entrada);
            end
            else begin
                read(entrada, c);
                haypalabra := not esblanco(c);
            end;
        end;
        while haypalabra do begin
            appcar(str, c);
            if eof(entrada) or eoln(entrada) then begin
                haypalabra := false;
            end
            else begin
                read(entrada, c);
                haypalabra := not esblanco(c);
            end;
        end;
    end;
```

La primera parte es el problema de ver si todos son cero, salvo porque estamos leyendo todos los blancos en lugar de buscar no-ceros en un array. La segunda parte es similar, pero opera al contrario y acumula los caracteres de la palabra en la palabra.

Aquí hemos utilizado varios procedimientos y funciones que nos hemos inventado.

Aquí tienes el programa completo por el momento:

```
1
2 {
3     Leer una cadena controladamente
4
5 }
6
7 program leestr(input, output);
```

```
 8
 9 const
10     LongStr = 50;
11     Tab: char = '    ';
12
13 type
14     TipoStr = record
15         cars: string[LongStr];
16         ncars: integer;
17     end;
18
19 procedure fatal(msg: string);
20 begin
21     writeln('error fatal: ', msg);
22     halt(1);
23 end;
24
25 procedure vaciarstr(var str: TipoStr);
26 begin
27     str.ncars := 0;
28 end;
29
30 procedure appcar(var str: TipoStr; c: char);
31 begin
32     if str.ncars = LongStr then begin
33         fatal('appcar: demasiados');
34     end;
35     str.ncars := str.ncars + 1;
36     str.cars[str.ncars] := c;
37 end;
38
39 function esblanco(c: char): boolean;
40 begin
41     result := (c = ' ') or (c = Tab);
42 end;
43
44 procedure leerstr(var entrada: Text; var str: TipoStr);
45 var
46     c: char;
47     haypalabra: boolean;
48 begin
49     vaciarstr(str);
50     haypalabra := false;
51     while not eof(entrada) and not haypalabra do begin
52         if eoln(entrada) then begin
53             readln(entrada);
54         end
55         else begin
```

```
56              read(entrada, c);
57              haypalabra := not esblanco(c);
58          end;
59      end;
60      while haypalabra do begin
61          appcar(str, c);
62          if eof(entrada) or eoln(entrada) then begin
63              haypalabra := false;
64          end
65          else begin
66              read(entrada, c);
67              haypalabra := not esblanco(c);
68          end;
69      end;
70  end;
71
72  procedure escrstr(str: TipoStr);
73  var
74      i: integer;
75  begin
76      for i := 1 to str.ncars do begin
77          write(str.cars[i]);
78      end;
79  end;
80
81  procedure escrstrln(str: TipoStr);
82  begin
83      escrstr(str);
84      writeln();
85  end;
86
87  var
88      str: TipoStr;
89  begin
90      leerstr(input, str);
91      escrstrln(str);
92  end.
```

Si ahora cambiamos el programa principal por el siguiente, tenemos el programa completo. El programa escribe una palabra por línea.

```
var
    str: TipoStr;
begin
    while not eof() do begin
        leerstr(input, str);
        if str.ncars > 0 then begin
```

```
            escrstrln(str);
        end;
    end;
end.
```

9.4. 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:

```
var
    str: TipoStr;
    max: TipoStr;
begin
    vaciarstr(max);
    while not eof() do begin
        leerstr(input, str);
        if str.ncars > max.ncars then begin
            max := str;
        end;
    end;
    if max.ncars = 0 then begin
        writeln('ninguna');
    end
    else begin
        escrstr(max);
        writeln();
    end;
end.
```

Tomamos como palabra mas larga una palabra vacía. 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. Resulta que todos los procedimientos y funciones utilizados son justo los que habíamos programado en el epígrafe anterior, así que no necesitamos hacer nada más. Compensa dividir los problemas en subproblemas.

9.5. ¿Por qué funciones de una línea?

Esto es, ¿Por qué molestarnos en escribir funciones como **`longstr`** para devolver la longitud de una palabra si en realidad no hacen casi nada? ¿No sería mucho mejor utilizar directamente **`str.ncars`** como hemos hecho?

La respuesta es simple, pero importante:

Hay que abstraer.

Si tenemos un tipo de datos y sus operaciones programadas con procedimientos y funciones, 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.

Si cambiamos el tipo de datos o la implementación de nuestro **`TipoStr`** tal y como lo tenemos en el programa anterior, el programa dejará de funcionar puesto que **`str.ncars`** quizá ni exista ya. En cambio, una función que podemos llamar como en **`longstr(str)`** podemos cambiarla para que siga funcionando perfectamente tras el cambio, y no tendremos que cambiar nada más.

9.6. 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 **`TipoStr`**, 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 Pascal 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 **`TipoRep`**, puesto que sirve para ver la frecuencia de aparición de una palabra.

```
TipoRep = record
    pal: TipoStr;
```

```
        nveces: integer;
    end;
```

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:

```
    LongCjto = 512;
```

Ahora ya nos podemos definir un *array* que matenga las repeticiones de palabras junto con un contador que nos diga 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 **TipoRep**. Por eso vamos a llamar a nuestro tipo **TipoCjtoReps**.

```
    TipoCjtoReps = record
        reps: array[1..LongCjto] of TipoRep;
        nreps: integer;
    end;
```

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.

```
    vaciarreps(reps);
    while not eof() do begin
        leerstr(input, str);
        if longitudstr(str) > 0 then begin
            insrep(reps, str);
        end;
    end;
```

Hemos vaciado nuestro conjunto de repeticiones (¡Que también nos hemos inventado!) con *vaciarreps* y utilizamos una operación *insrep* para insertar una repetición 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 entonces *insrep* 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:

```
    str := maxrep(reps);
    if longitudstr(str) = 0 then begin
        writeln('ninguna');
    end
    else begin
        escrstrln(str);
```

```
    end;
```

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.

```
procedure insrep(var cjto: TipoCjtoReps; pal: TipoStr);
var
    i: integer;
    encontrado: boolean;
begin
    i := 1;
    encontrado := false;
    while (i <= cjto.nreps) and not encontrado do begin
        if igualstr(cjto.reps[i].pal, pal) then begin
            incrrep(cjto.reps[i]);
            encontrado := true;
        end
        else begin
            i := i + 1;
        end;
    end;
    if not encontrado then begin
        nuevarep(cjto, pal);
    end;
end;
```

Siempre es lo mismo. Nos inventamos todo lo que necesitemos. Ahora necesitamos una operación **`incrrep`** y otra **`nuevarep`**. Nos las vamos inventando una tras otra a no ser que lo que tengamos que hacer sea tan simple que queramos escribirlo *in-situ*.

El procedimiento para buscar es similar la los que vimos antes, salvo por un detalle: para comparar palabras no podemos comparar directamente. Necesitamos una función **`igualstr`** 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.

El programa completo quedaría como sigue:

```
 1
 2 {
 3     Escribir palabra mas repetida
 4
 5 }
 6
 7 program maxreps;
 8
 9 const
10     LongStr = 80;
11     LongCjto = 512;
12     Tab: char = '    ';
13
14 type
15     TipoStr = record
16         cars: string[LongStr];
17         ncars: integer;
18     end;
19
20     TipoRep = record
21         pal: TipoStr;
22         nveces: integer;
23     end;
24
25     TipoCjtoReps = record
26         reps: array[1..LongCjto] of TipoRep;
27         nreps: integer;
28     end;
29
30 procedure fatal(msg: string);
31 begin
32     writeln('error fatal: ', msg);
33     halt(1);
34 end;
35
36 procedure vaciarstr(var str: TipoStr);
37 begin
38     str.ncars := 0;
39 end;
40
41 function longitudstr(str: TipoStr): integer;
42 begin
43     result := str.ncars;
44 end;
45
46 procedure appcar(var str: TipoStr; c: char);
47 begin
```

```
48      if str.ncars = LongStr then begin
49          fatal('appcar: demasiados');
50      end;
51      str.ncars := str.ncars + 1;
52      str.cars[str.ncars] := c;
53  end;
54
55  function esblanco(c: char): boolean;
56  begin
57      result := (c = ' ') or (c = Tab);
58  end;
59
60  procedure leerstr(var entrada: Text; var str: TipoStr);
61  var
62      c: char;
63      haypalabra: boolean;
64  begin
65      vaciarstr(str);
66      haypalabra := false;
67      while not eof(entrada) and not haypalabra do begin
68          if eoln(entrada) then begin
69              readln(entrada);
70          end
71          else begin
72              read(entrada, c);
73              haypalabra := not esblanco(c);
74          end;
75      end;
76      while haypalabra do begin
77          appcar(str, c);
78          if eof(entrada) or eoln(entrada) then begin
79              haypalabra := false;
80          end
81          else begin
82              read(entrada, c);
83              haypalabra := not esblanco(c);
84          end;
85      end;
86  end;
87
88  procedure escrstr(str: TipoStr);
89  var
90      i: integer;
91  begin
92      for i := 1 to str.ncars do begin
93          write(str.cars[i]);
94      end;
95  end;
```

```
96
97 function igualstr(str1, str2: TipoStr): boolean;
98 var
99     i: integer;
100     igual: boolean;
101 begin
102     if longitudstr(str1) <> longitudstr(str2) then begin
103         result := false;
104     end
105     else begin
106         igual := true;
107         i := 1;
108         while (i <= longitudstr(str1)) and igual do begin
109             if str1.cars[i] <> str2.cars[i] then begin
110                 igual := false;
111             end
112             else begin
113                 i := i + 1;
114             end;
115         end;
116         result := igual;
117     end;
118 end;
119
120 procedure escrstrln(str: TipoStr);
121 begin
122     escrstr(str);
123     writeln();
124 end;
125
126 procedure vaciarreps(var cjto: TipoCjtoReps);
127 begin
128     cjto.nreps := 0;
129 end;
130
131 procedure incrrep(var rep: TipoRep);
132 begin
133     rep.nveces := rep.nveces + 1;
134 end;
135
136 procedure nuevarep(var cjto: TipoCjtoReps; pal: TipoStr);
137 begin
138     if cjto.nreps = LongCjto then begin
139         fatal('conjunto demasiado largo');
140     end;
141     cjto.nreps := cjto.nreps + 1;
142     cjto.reps[cjto.nreps].pal := pal;
143     cjto.reps[cjto.nreps].nveces := 1;
```

```
144 end;
145
146 procedure insrep(var cjto: TipoCjtoReps; pal: TipoStr);
147 var
148     i: integer;
149     encontrado: boolean;
150 begin
151     i := 1;
152     encontrado := false;
153     while (i <= cjto.nreps) and not encontrado do begin
154         if igualstr(cjto.reps[i].pal, pal) then begin
155             incrrep(cjto.reps[i]);
156             encontrado := true;
157         end
158         else begin
159             i := i + 1;
160         end;
161     end;
162     if not encontrado then begin
163         nuevarep(cjto, pal);
164     end;
165 end;
166
167 function maxrep(var cjto: TipoCjtoReps): TipoStr;
168 var
169     i: integer;
170     cual: TipoRep;
171 begin
172     vaciarstr(cual.pal);
173     cual.nveces := 0;
174     for i := 1 to cjto.nreps do begin
175         if cjto.reps[i].nveces > cual.nveces then begin
176             cual := cjto.reps[i];
177         end;
178     end;
179     result := cual.pal;
180 end;
181
182 var
183     str: TipoStr;
184     reps: TipoCjtoReps;
185 begin
186     vaciarreps(reps);
187     while not eof() do begin
188         leerstr(input, str);
189         if longitudstr(str) > 0 then begin
190             insrep(reps, str);
191         end;
```

```
192     end;
193     str := maxrep(reps);
194     if longitudstr(str) = 0 then begin
195         writeln('ninguna');
196     end
197     else begin
198         escrstrln(str);
199     end;
200 end.
```

9.7. Lectura de enteros

En ocasiones desearemos leer un entero de forma controlada, o cualquier otro tipo de datos. Una solución es utilizar la lectura controlada de palabras que hemos visto y, tras leer una palabra, comprobar si corresponde al texto de algo que deseamos leer o no.

Por ejemplo, para leer un entero podemos utilizar **`leerstr`** y después intentar convertir la cadena de cacteres que hemos leído a un entero o informar de que no es posible la conversión.

Pascal posee el procedimiento **`val`** que calcula el valor numérico de un *string* informando cuál es la posición del primer carácter del *string* que no ha sido posible utilizar como parte del número. Por ejemplo,

```
val(s, n, pos);
```

hace que **`n`** sea el valor del entero cuyo texto está en **`s`** y deja en **`pos`** un cero si **`s`** contenía "**`patata`**" (puesto que no se ha utilizado ningún carácter al no ser posible extraer ningún número).

Teniendo esto en cuenta, el siguiente programa escribe los valores enteros que aparecen en la entrada, ignorando el resto de palabras. Hemos utilizado una función **`strtostring`** que convierte nuestro tipo para palabras en un *string* que podemos utilizar con **`val`**.

```
 1
 2 {
 3     Leer enteros controladamente
 4
 5 }
 6
 7 program leerint;
 8
 9 const
10     LongStr = 80;
11     Tab: char = '    ';
```

```
12
13 type
14     TipoStr = record
15         cars: string[LongStr];
16         ncars: integer;
17     end;
18
19 procedure fatal(msg: string);
20 begin
21     writeln('error fatal: ', msg);
22     halt(1);
23 end;
24
25 procedure vaciarstr(var str: TipoStr);
26 begin
27     str.ncars := 0;
28 end;
29
30 procedure appcar(var str: TipoStr; c: char);
31 begin
32     if str.ncars = LongStr then begin
33         fatal('appcar: demasiados');
34     end;
35     str.ncars := str.ncars + 1;
36     str.cars[str.ncars] := c;
37 end;
38
39 function esblanco(c: char): boolean;
40 begin
41     result := (c = ' ') or (c = Tab);
42 end;
43
44 procedure leerstr(var entrada: Text; var str: TipoStr);
45 var
46     c: char;
47     haypalabra: boolean;
48 begin
49     vaciarstr(str);
50     haypalabra := false;
51     while not eof(entrada) and not haypalabra do begin
52         if eoln(entrada) then begin
53             readln(entrada);
54         end
55         else begin
56             read(entrada, c);
57             haypalabra := not esblanco(c);
58         end;
59     end;
```

```
60      while haypalabra do begin
61          appcar(str, c);
62          if eof(entrada) or eoln(entrada) then begin
63              haypalabra := false;
64          end
65          else begin
66              read(entrada, c);
67              haypalabra := not esblanco(c);
68          end;
69      end;
70  end;
71
72  procedure escrstr(str: TipoStr);
73  var
74      i: integer;
75  begin
76      for i := 1 to str.ncars do begin
77          write(str.cars[i]);
78      end;
79  end;
80
81  procedure escrstrln(str: TipoStr);
82  begin
83      escrstr(str);
84      writeln();
85  end;
86
87  function strtostring(str: TipoStr): string;
88  var
89      i: integer;
90      s: string;
91  begin
92      setlength(s, str.ncars);
93      for i := 1 to str.ncars do begin
94          s[i] := str.cars[i];
95      end;
96      result := s;
97  end;
98
99  procedure leerint(var entrada: text;
100              var n: integer; var ok: boolean);
101 var
102     str: TipoStr;
103     s: string;
104     pos: integer;
105 begin
106     ok := not eof();
107     if ok then begin
```

```
108         leerstr(entrada, str);
109         s := strtostring(str);
110         val(s, n, pos);
111         ok := pos = 0;
112     end;
113 end;
114 
115 var
116     n: integer;
117     ok: boolean;
118 begin
119     repeat
120         leerint(input, n, ok);
121         if ok then begin
122             writeln(n);
123         end;
124     until eof();
125 end.
```

9.8. 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.

Índice