Tipos estructurados de datos

Tablas y tuplas

Ramon Vilanova i Arbós
Raymond Lagonigro i Bertran

PID_00149894

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción 5

Objetivos 6

1. Introducción y motivación. Estructuración de datos 7
1.1. Motivación de las tablas 8
1.2. Motivación de las tuplas 9

2. Tablas 11
2.1. Declaración 11
2.2. Operaciones elementales de acceso 12
2.2.1. Asignación y consulta 14
2.2.2. Lectura y escritura 14
2.3. Otros tipos de tablas 17
2.3.1. Tablas de varias dimensiones 17
2.3.2. Tablas de tablas 18
2.4. Acceso secuencial a una tabla 19
2.4.1. Esquema de recorrido aplicado a tablas 19
2.4.2. Esquema de búsqueda aplicado a tablas 24
2.4.3. Ejemplos de los esquemas de recorrido y búsqueda 27
2.5. Acceso directo a una tabla 32
2.5.1. Importancia de los datos ordenados. Búsqueda dicotómica 32
2.5.2. Ordenación por selección 34

3. Tuplas 37
3.1. Declaración 37
3.2. Operaciones elementales de acceso 39
3.2.1. Asignación y consulta 40
3.2.2. Lectura y escritura por los canales de entrada/salida estándar 40
3.3. Otros tipos de tuplas 42
3.3.1. Tuplas de tuplas 42
3.3.2. Tablas y tuplas 44

Resumen 47

Ejercicios de autoevaluación 53

Solucionario 56

Glosario 67

Bibliografía 67

Introducción

Los tipos de datos que hemos visto en los módulos anteriores eran suficientes para representar y tratar la información de los problemas que se han ido planteando, pero, como veremos en este módulo, hay problemas que requieren objetos más estructurados, que se adapten mejor a la información que será necesario representar en cada algoritmo concreto.

De hecho, lo que veremos será cómo construir, a partir de los tipos de datos básicos que hemos visto en el módulo "Introducción a la algorítmica", otros más estructurados, que, en algunos casos por cuestiones de eficiencia y en otros por cuestiones de legibilidad, nos permitirán solucionar mejor los problemas que se nos plantearán.

Los dos nuevos tipos de datos que estudiaremos en este módulo son las tablas y las tuplas. Tanto las tablas como las tuplas sirven para agrupar tipos de datos simples.

Las tuplas permiten agrupar datos diferentes para formar un objeto mayor que los incluye. Esta agrupación se lleva a cabo básicamente por cuestiones de legibilidad y para facilitar el diseño. Tengamos en cuenta que, en nuestros algoritmos, tenemos que representar datos del mundo real para resolver problemas del mundo real. Las tuplas nos permitirán diseñar unos objetos para nuestros algoritmos que se adapten al máximo a los datos que representan, de manera que el tratamiento sea mucho más evidente.

Los elementos que agrupamos en las tablas, tienen que ser todos del mismo tipo. La agrupación se lleva a cabo para poder hacer tratamientos más eficientes de la información.

Veremos que en el tratamiento de tablas son válidos los esquemas que hemos aprendido en el módulo "Tratamiento secuencial: búsqueda y recorrido". Y por tanto, veremos cómo aplicar estos esquemas.

Veremos también que las tablas permiten acceder a sus elementos de forma directa. El acceso directo es una de las características importantes de las tablas que no hemos visto todavía en módulos anteriores. Este tipo de acceso nos permitirá realizar unos procedimientos de ordenación y búsqueda sólo posibles con las tablas.

Objetivos

Después de haber estudiado este módulo, tendréis que haber alcanzado los siguientes objetivos:

- Darse cuenta de la necesidad de utilizar tablas para resolver un problema concreto de forma eficiente.
- Saber diseñar adecuadamente las tablas en los algoritmos.
- Aplicar correctamente los esquemas de recorrido y búsqueda para el tratamiento de tablas.
- Entender la importancia del acceso directo.
- Utilizar las tuplas para facilitar la comprensión de los algoritmos.
- Diseñar nuevos tipos de datos, mediante tuplas y tablas, adecuados a cada problema concreto.

1. Introducción y motivación. Estructuración de datos

Como hemos ido observando en los módulos anteriores, para resolver un problema diseñamos un algoritmo que manipula un conjunto de objetos. Estos objetos nos permiten representar los datos del problema dentro de nuestro algoritmo. Hasta ahora hemos visto objetos de tipo elemental: enteros, reales, caracteres y booleanos. Hemos visto también un mecanismo, el constructor de tipos, que nos permite definir nuevos tipos para cada algoritmo, y hemos aprendido a utilizarlo para crear tipos enumerados.

Consultad el apartado 1 del módulo didáctico "Introducción a la algorítmica".

A medida que vayamos avanzando en el aprendizaje de las técnicas de programación, nos iremos encontrando con problemas que hay que resolver, cuya complejidad exige que nuestros algoritmos puedan manipular tipos de datos estructurados para definir objetos que se adapten o se parezcan más a los datos del problema, o bien para definir objetos que nos permitan hacer un tratamiento más eficiente de la información.

Los tipos de datos estructurados...

... permiten agrupar o relacionar tipos de datos elementales.

Estructurar los datos quiere decir agruparlos de una forma. Esta agrupación nos ofrecerá las siguientes posibilidades:

- Hacer los algoritmos más eficientes: la forma de estructurar los datos nos permitirá ciertos tratamientos más potentes.

- Hacer los algoritmos más legibles, gracias a la expresividad de los tipos estructurados.

Podemos clasificar los tipos de datos estructurados según varios parámetros:

1) El tipo de elementos que agrupamos:
- Todos del mismo tipo: estructura homogénea.
- De tipos diferentes: estructura heterogénea.

2) La forma como se accede a los elementos:
- Acceso secuencial: el acceso a los elementos se consigue en el mismo orden según el que están colocados. Para obtener el elemento de la i-ésima posición, antes tenemos que obtener los $i - 1$ elementos anteriores.
- Acceso directo: podemos obtener directamente cualquiera de los elementos de la estructura.

3) El número de elementos que puede contener:
- Tamaño fijo: el número de elementos se define en la declaración y no puede variar en tiempo de ejecución.
- Tamaño variable: en tiempo de ejecución puede variar el número de elementos.

Las diferentes combinaciones de estas características darían lugar a muchos tipos de datos estructurados. Incluso podríamos definir tipos diferentes con las mismas características. Se da el caso de ciertos tipos de datos estructurados que sólo existen en lenguajes de programación muy concretos. Puesto que pretendemos exponer nuestros criterios con la mayor generalidad posible, en este módulo nos centraremos en dos tipos de datos estructurados que podemos encontrar en prácticamente todos los lenguajes de alto nivel: las tablas y las tuplas.

El tipo **tabla** es un tipo de datos estructurado, homogéneo, de acceso directo y dimensión fija.

El tipo **tupla** es un tipo de datos estructurado, heterogéneo, de acceso directo y dimensión fija.

A pesar de que...

... las tablas y las tuplas se diferencian sólo en la homogeneidad o no del tipo de sus elementos, como veremos, el tratamiento y la finalidad son muy diferentes.

1.1. Motivación de las tablas

Las tablas nos permiten agrupar elementos del mismo tipo, bajo un único nombre, en un orden concreto. En las tablas, el elemento que diferencia un elemento de otro es la posición que ocupa cada uno dentro de la tabla. Es decir, el orden de los elementos es lo que los define.

Nos referimos a cada elemento indicando la posición que éste ocupa dentro de la tabla. Esta característica hace que las tablas resulten ideales para ciertos tipos de tratamientos que, si no pudieran utilizarlas, serían muy deficientes.

Por ejemplo, imaginemos que tenemos que escribir un algoritmo que cuente las vocales que aparecen en un texto:

Utilizar tablas reduce el número de líneas de código que hay que escribir.

Sin utilizar tablas necesitaremos cinco variables (*cuentA*, *cuentE*, *cuentI*, *cuentO*, *cuentU*), de tipo **entero**, y cada una nos permitirá contar una vocal. Antes de poder empezar a contar tendremos que inicializar estas variables a 0:

```
cuentA := 0;
cuentE := 0;
cuentI  := 0;
cuentO := 0;
cuentU := 0;
```

En cambio, si utilizamos tablas, necesitaremos una con cinco elementos de tipo **entero**, y cada uno contará una vocal. En este caso, la inicialización de los elementos de la tabla será:

Para i := 1 **hasta** 5 **hacer** cuent[i] := 0; **fpara**

Fijaos en que, si en lugar de contar vocales contamos todas las letras, en el primer caso necesitaremos tantas variables como letras diferentes contiene el alfabeto, así como el mismo número de líneas de código para llevar a cabo la inicializa-

ción. En cambio, en el segundo caso sólo es necesario cambiar el número de elementos de la tabla y el número de iteraciones del bucle.

1.2. Motivación de las tuplas

En el caso de las tuplas, la agrupación no se efectúa por razones de eficiencia sino de claridad, definiendo tipos de datos que se adapten al máximo a los datos del problema que hay que tratar.

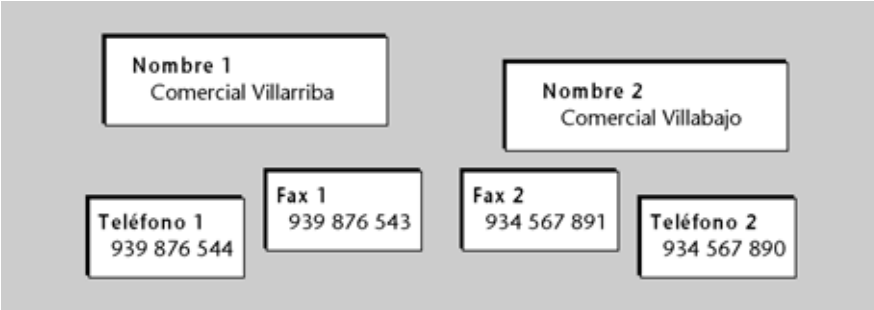
Las tuplas agrupan elementos que pueden presentar tipos diferentes pero que se refieren todos a un mismo objeto. Cada uno de estos elementos tiene un nombre diferente, de manera que identificamos los elementos con el nombre de la tupla y el nombre del elemento. El acceso es directo, ya que accedemos a cada elemento por su nombre.

El orden de los elementos dentro de la tupla es indiferente.

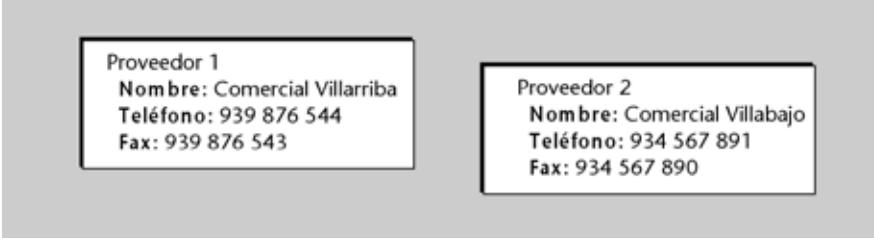
Veamos el interés de las tuplas mediante el siguiente ejemplo:

Imaginemos una oficina que guarda información sobre sus dos únicos proveedores. En concreto tiene de cada uno el nombre, el teléfono y el fax.

Cada una de estas informaciones está escrita en un trozo de papel con un título que indica qué es lo que hay en el papel, y si pertenece al proveedor 1 o al proveedor 2:



Es fácil darse cuenta de que esta forma de guardar la información no es la más adecuada, ya que provoca una dispersión de los datos. Sería mucho más indicado tratar al proveedor como un bloque de información incluida en unas fichas, de manera que toda la información de un proveedor estuviese en el mismo papel:



Probablemente todos hubiésemos escogido esta segunda opción para guardar la información.

Con los tipos de datos que hemos visto hasta ahora, no disponemos de ninguna herramienta que nos permita utilizar esta representación en un algoritmo;

por tanto, si diseñamos un algoritmo que utilice la información de los dos proveedores del ejemplo, las variables que tendremos que utilizar serán:

```
var
    nombreProveedor1, nombreProveedor2: tabla [15] de caracter;
    telefonoProveedor1, faxProveedor1: entero;
    telefonoProveedor2, faxProveedor2: entero
fvar
```

Supongamos que el tipo entero nos permite guardar números de nueve cifras.

Es importante que tengamos en cuenta que se nos plantea el mismo problema de dispersión que vimos en el ejemplo anterior.

Las tuplas...

... nos permiten agrupar varios objetos de tipos diferentes para formar uno más genérico que los englobe a todos.

Necesitamos un mecanismo que nos permita definir un tipo de datos nuevo, específico para este algoritmo, que podríamos llamar *proveedor*, y que tendría que guardar los tres datos, nombre, teléfono y fax correspondientes a un proveedor. Este tipo de datos será una tupla:

```
tipo
        proveedor =  tupla
                        nombre: tabla [15] de carácter;
                        telefono, fax: entero;
                     ftupla
ftipo
```

2. Tablas

2.1. Declaración

Hemos comentado que una tabla es un tipo estructurado de datos en el que agrupamos un conjunto de variables del mismo tipo de acuerdo con un orden determinado.

En el lenguaje algorítmico, la declaración de una variable de tipo **tabla** obedece a la siguiente sintaxis:

```
var
        nombre: tabla [tamaño] de tipoBase
fvar
```

En esta declaración tenemos presentes las tres partes que forman una tabla; es decir, los elementos que estamos agrupando, bajo qué nombre haremos referencia a la agrupación y cómo accederemos a cada uno de los elementos de esta agrupación:

1) *tipoBase*: es el tipo de los elementos que estamos agrupando. Así, todos los elementos de la tabla serán de este mismo tipo. Sin embargo, remarcamos que este *tipoBase* puede ser cualquier tipo ya definido, incluso un tipo estructurado.

2) *nombre*: éste es el identificador de la variable tipo tabla. Con este nombre sólo haremos referencia a todos los elementos de la tabla. De esta forma, en lugar de tener una para cada elemento, tenemos una única variable. Cada uno de los elementos de la tabla será accesible individualmente mediante una expresión entre corchetes que nos indicará la posición que ocupa el elemento dentro de la misma.

Recordad lo que comentamos en el módulo de "Introducción a la algorítmica" referente a los nombres de las variables.

3) *tamaño*: Aquí indicamos la cantidad de elementos del *tipoBase* que contendrá la tabla. Por tanto, como cantidad que es, tiene que ser un número positivo. No tiene sentido agrupar –4 caracteres o 6.5 enteros.

De este modo, la tabla constará de *tamaño* elementos del tipo *tipoBase*. Ya que, tal como hemos comentado, la tabla es una disposición ordenada de elementos del mismo tipo, hablaremos del primer elemento, el segundo elemento, y así hasta el *tamaño*-ésimo elemento.

Veamos, a continuación, algunos ejemplos de declaración de tablas:

Ejemplo 1

```
var
    t: tabla [10] de caracter;
fvar
```

donde declaramos una tabla *t* con 10 elementos de tipo **caracter**.

Una aplicación...

… en la que podría utilizarse esta tabla sería la de contar las apariciones de cada carácter en un texto. Podemos asociar la primera posición con el número de apariciones del carácter 'a', la segunda con el número de apariciones del carácter 'b', etc.

Ejemplo 2

```
var
    apariciones: tabla [26] de entero;
fvar
```

donde declaramos un tabla de nombre *apariciones* que contendrá 26 elementos de tipo **entero**.

Ejemplo 3

Supongamos que nos interesa guardar la recaudación diaria de un cine. Lo podríamos hacer, utilizando tablas, de la siguiente forma:

```
var
    recaudacionCine: tabla [7] de entero;
fvar
```

donde tenemos una tabla de siete elementos (uno para cada día de la semana), cada uno de tipo **entero**, que representarán la recaudación del día correspondiente. Así, asociaríamos la primera posición con la recaudación del lunes, la segunda con la del martes, etc.

2.2. Operaciones elementales de acceso

Ya sabemos cómo introducir, mediante el lenguaje algorítmico, variables de tipo tabla en nuestros algoritmos. En el uso de variables de este nuevo tipo debemos tener claro que no podemos realizar ninguna operación ni con la tabla entera ni con partes de la misma. Lo que hacemos es trabajar con cada uno de los elementos como si se tratase de una variable del tipo *tipoBase*.

Consultad el apartado referente a objetos en el mòdulo "Introducción a la Algorítmica".

Cuando hablamos de operar con tablas nos estamos refiriendo a las operaciones asociadas a los diferentes tipos elementales de datos que ya conocemos, es decir, no podemos comparar directamente dos tablas mediante el operador '=', tal y como lo haríamos para comparar, por ejemplo, dos enteros. Sin embargo, en el ámbito de acciones y funciones sí podemos utilizar el tipo tabla como un parámetro más de éstas, ya sea de entrada o de salida. Es necesario, por tanto, no confundir el hecho de no poder operar directamente con toda la tabla con la posibilidad de pasarla como un parámetro a una acción o función.

Así pues, tenemos que ver primero cómo hay que hacer referencia a cada uno de los elementos de la tabla. Esto nos permitirá introducir, después, cuáles son las operaciones elementales de acceso a una tabla. Mediante estas operaciones podremos sacar partido de las tablas.

Referenciar un elemento concreto de una tabla es muy sencillo si recordamos que hemos presentado la tabla como una agrupación de elementos: bajo un determi-

nado identificador, junto con un índice que establece una ordenación de estos elementos dentro del grupo.

Con el fin de referenciar un elemento concreto de una tabla, especificaremos el nombre de la tabla y, entre corchetes, la posición que ocupa el elemento de acuerdo con el índice. Es decir:

nombreTabla[expresión]

donde *nombreTabla* hace referencia al nombre de la tabla y *expresión* es una expresión que evalúa un número entero positivo que tiene que estar dentro de los límites marcados por la dimensión de la tabla.

Ejemplo 1

Si tenemos la siguiente declaración de tabla

```
var
    t: tabla [10] de entero;
fvar
```

Podremos hacer, por ejemplo:

t[3]	Estaríamos accediendo al elemento que ocupa la tercera posición.
t[k]	Con k una variable entera y $1 \leq k \leq 10$.
t[$2 * j - k$]	Con j y k variables enteras y $1 \leq 2 * j - k \leq 10$.

y no sería correcto hacer

t[–5]	Estaríamos fuera de los límites permitidos por el índice.

El hecho de referenciar un elemento cualquiera de la tabla quiere decir que el acceso para operar con los mismos es directo; propiedad, tal y como veremos más adelante, importante de las tablas.

Que el acceso a un elemento sea directo...

... quiere decir que para acceder a este elemento no es necesario acceder a los elementos anteriores.

Ejemplo 2

Por la declaración que ya conocemos,

```
var
    apariciones: tabla [26] de entero;
fvar
```

apariciones[1] hará referencia al primer elemento de la tabla, *apariciones*[2] al segundo, *apariciones*[3] al tercero, y así sucesivamente.

2.2.1. Asignación y consulta

En una **asignación**, lo que hacemos es, precisamente, asignar un valor a un elemento concreto de la tabla. Así, por ejemplo, con

t[5]:= 11;

asignamos el valor 11 a la posición 5 de la tabla *t*.

El hecho de que no podamos operar con toda la tabla entera implica, entre otros aspectos, que una asignación de la forma t := 11; no tendría ningún sentido, ya que a ambos lados del operador de asignación debemos tener variables del mismo tipo y, en este caso, una cosa es una tabla de enteros y otra un entero.

Por lo que respecta a la **consulta** del valor que tiene un elemento de la tabla, ésta es inmediata una vez se ha especificado la posición que ocupa este elemento dentro de la misma. Este valor se puede utilizar en un contexto cualquiera de la misma forma que lo haríamos con una variable del tipo *tipoBase*. Así, si *x* es una variable entera, podríamos hacer la asignación siguiente:

x := 3 ∗ t[4] − 10;

donde, con *t*[4], obtenemos el valor de la posición 4 de la tabla.

2.2.2. Lectura y escritura

Estas dos operaciones las podríamos considerar, de hecho, como una aplicación directa de la asignación y consulta expuestas. En efecto, podemos leer un dato de un dispositivo de entrada y guardarlo en una posición de una tabla y escribir en un dispositivo de salida el valor de un elemento de la tabla.

Estos ejemplos....

... utilizan, ya que la tabla es de enteros, las acciones básicas de entrada y salida para el tipo entero. De la misma manera utilizaríamos, si la tabla fuese de caracteres, *escribirCaracter*, *leerCaracter*, etc.

Ejemplos de escritura y lectura	
escribirEntero(t[2])	Escribe en el dispositivo de salida el valor contenido en la posición 2 de la tabla *t*.
escribirEntero(t)	No tiene sentido, ya que implica operar con la tabla entera (provocaría error).
t[4] := **leerEntero**( )	Lee un dato del dispositivo de entrada y lo asigna a la posición 4 de la tabla *t*.
t := **leerEntero**( )	No tiene sentido, ya que implica operar con la tabla entera (provocaría error).

Ejemplo

Dada una secuencia de caracteres en mayúsculas y acabada en punto, que leeremos del canal de entrada, se trata de obtener la frecuencia de aparición de

cada letra del alfabeto. Esta frecuencia será el número de veces que aparece la letra dentro de la secuencia dividido por el número total de caracteres.

Fijémonos en que, para contar la frecuencia de aparición de cada letra, necesitaríamos tantos contadores o variables como letras tiene el alfabeto. Estas variables, sin embargo, serán todas del mismo tipo, lo que nos sugiere la posibilidad de agruparlas y utilizar una tabla.

Especifiquemos primero este algoritmo:

```
frec: tabla [26] de real
{ Pre: leeremos por el canal de entrada una secuencia de caracteres, en mayúsculas, acabada en punto }
frecLetras
{ Post: frec nos proporciona la frecuencia de aparición de los caracteres 'A' hasta 'Z', de manera que frec[1] se corresponde con la del carácter 'A', frec[2] con la del carácter 'B', y así sucesivamente hasta la frec[26], que se corresponde con la del carácter 'Z' }
```

Después, lo desarrollamos:

```
algoritmo frecLetras
    var
        frec: tabla [26] de real;
        ndc, i, codigoBase: entero;
        c: caracter
    fvar
    c := leerCaracter(); ndc := 0;
    para i:= 1 hasta 26 hacer
        frec[i]:= 0.0;
    fpara
    codigoBase := caracterACodigo('A');
    mientras c ≠ '.' hacer
        i := caracterACodigo(c) – codigoBase + 1;
        frec[i] := frec[i] + 1.0;
        ndc := ndc + 1;
        c := leerCaracter()
    fmientras
    para i := 1 hasta 26 hacer
        frec[i] := frec[i] / enteroAReal(ncd)
        escribirCaracter(codigoACaracter(codigoBase + i – 1));
        escribirCaracter(':');
        escribirReal(frec[i])
    fpara
falgoritmo
```

Fijémonos en que...

... nos hemos basado, para determinar el valor del índice, en que los códigos de los diferentes caracteres son correlativos. Así, si recordamos el módulo de "Introducción a la Algorítmica", tenemos que el código de la 'A' es el 65, el de la 'B' el 66, y así sucesivamente.

De este modo, por ejemplo, al carácter 'C' le corresponderá la tercera posición de la tabla o, lo que es lo mismo, un valor del índice igual a 3:

$$i := 67 - 65 + 1$$

Fijémonos en que, tal y como presentamos las operaciones de lectura y escritura, éstas son específicas para el tipo de datos con los que estamos operando. Así, las dos situaciones anteriores en las que se provocaría un error ya no tendrán sentido de buen principio si pensamos que no tenemos una función correspondiente de lectura/escritura asociada al tipo tabla. Ahora bien, ¿no hemos dicho que una tabla sí se puede utilizar como parámetro de una acción y/o función? Esto nos lleva a la posibilidad de escribir nosotros mismos las correspondientes funciones de lectura/escritura asociadas a las tablas que definimos en nuestros algoritmos. Veamos un ejemplo.

Ejemplo

Escribiremos un algoritmo que, a partir de una tabla de enteros, nos calcule los cuadrados de sus elementos.

Lo que haremos es escribir el algoritmo a partir de acciones que nos ayuden a estructurar las operaciones de lectura y escritura, así como una función que realice la operación de calcular los cuadrados. Fijémonos en que, de esta manera, estamos separando en nuestro algoritmo tareas suficientemente diferenciadas como la obtención, procesamiento y visualización de los datos.

Especificamos primero las acciones y funciones que componen este algoritmo

```
accion llenarTabla(entsal p: t)
{ Pre: p es una tabla de M enteros }
{ Post: la tabla p se ha llenado con M valores leídos del canal de entrada }

accion escribirTabla(ent p: t)
{ Pre: p es una tabla de M enteros }
{ Post: los M valores de la tabla p se han escrito en el canal de salida }

funcion cuadradosTabla(ent p: t): t
{ Pre: p = P es una tabla de M enteros }
{ Post: cada elemento de la tabla devuelta por la función contiene el cuadrado del correspondiente elemento de P, que ocupa la misma posición }
```

Una vez especificadas las acciones y funciones ya podemos desarrollar el algoritmo:

```
algoritmo cuadrados
    const M: entero = 10; fconst
    tipo t = tabla [M] de entero; ftipo
    var
        tablaI, tablaF: t;
    fvar
    llenarTabla(tablaI);
    tablaF := cuadradosTabla(tablaI);
    escribirTabla(tablaF);
falgoritmo
accion llenarTabla(entsal p: t)
var
    i: entero;
fvar
    para i := 1 hasta M hacer
            p[i] := leerEntero()
    fpara
faccion
accion escribirTabla(ent p: t)
var
    i: entero;
fvar
    para i := 1 hasta M hacer
        escribirEntero(p[i])
    fpara
faccion
funcion cuadradosTabla(p: t) : t
        var
            i: entero;
            nuevap: t;
        fvar
    para i := 1 hasta M hacer
        nuevap[i] := p[i] * p[i]
    fpara
    retorna nuevap
ffuncion
```

2.3. Otros tipos de tablas

2.3.1. Tablas de varias dimensiones

Las tablas que hemos introducido hasta ahora las podemos imaginar como una representación de vector. En el caso concreto de la tabla que podíamos utilizar para guardar la recaudación del cine en los diferentes días de la semana, tendríamos:

Representación en forma de vector

35	12	23	56	87	90	75	Vector de valores
1a	2a	3a	4a	5a	6a	7a	Posición en el vector
1	2	3	4	5	6	7	Valor del índice
Lunes	Martes	Miércoles	Jueves	Viernes	Sábado	Domingo	

donde accedemos a uno de los elementos especificando, mediante el correspondiente valor del índice, la posición que ocupa dentro de este vector.

Si imaginamos ahora que este cine tiene tres salas y que queremos, igualmente, distinguir la recaudación en cada sala, entonces nos hará falta guardar para todos los días de la semana tres valores. Una posible manera de representar esta información es basándose en una matriz, una generalización directa de la representación en forma de vector anterior. Así, llegaremos a una representación de la forma:

Representación en forma de matriz

35	12	23	56	87	90	75	Sala 1
28	10	20	50	84	95	80	Sala 2
40	15	30	50	90	110	90	Sala 3
Lunes	Martes	Miércoles	Jueves	Viernes	Sábado	Domingo	

Ahora, con el objetivo de acceder a la recaudación de una sala en un día de la semana concreto, será necesario especificar, lógicamente, dos valores que nos situarán en la fila y columna correspondientes.

Efectivamente, hemos llegado a una tabla de dos dimensiones en la que necesitamos dos valores del índice para poder referenciar un elemento.

La **declaración de una tabla bidimensional** la haríamos como:

var
nombre: **tabla** [*tamaño*1, *tamaño*2] **de** *tipoBase*
fvar

Ejemplo

Una tabla para guardar la recaudación por sala durante los días de una semana se podría declarar como:

```
var
    recaudacionCine: tabla [7, 3] de entero;
fvar
```

Para asignar, por ejemplo, una recaudación de 84 el viernes en la sala 2, lo haríamos de la siguiente manera:

```
recaudacionCine [5, 2] := 84;
```

Sin embargo, no tenemos que limitarnos a dos dimensiones o, si queremos, a sólo dos valores del índice. En efecto, una tabla *n*-dimensional será una tabla en la que utilicemos *n* valores del índice para acceder a un elemento.

La **declaración de una tabla *n*-dimensional** la haríamos como:

var
nombre: **tabla** [*tamaño*1, *tamaño*2, … , *tamaño n*] **de** *tipoBase*;
fvar

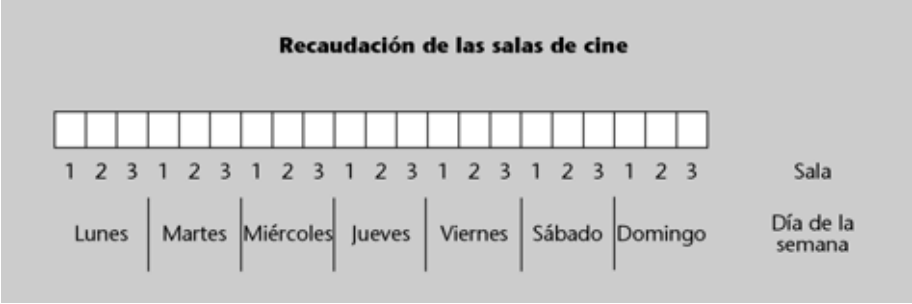
2.3.2. Tablas de tablas

El hecho de que el tipo base de los elementos que agrupamos en una tabla pueda ser un tipo cualquiera y, en especial, un tipo estructurado, nos ofrece la posibilidad de pensar en una tabla donde cada uno de sus elementos es, a su vez, también una tabla.

Veámoslo aplicado al ejemplo del apartado anterior en el que representábamos la recaudación de las tres salas de un cine durante los días de una semana. En efecto, lo que nos interesa tener para cada día de la semana es, de hecho, una tabla de tres elementos (correspondientes a los tres valores de las tres recaudaciones). De modo que tendríamos la siguiente declaración:

```
var
    recaudacionCine: tabla [7] de tabla [3] de entero;
fvar
```

En este caso, la representación que se correspondería con esta tabla ya no sería la equivalente a una matriz, sino una como ésta:



Tenemos un vector de siete elementos donde cada elemento es un vector de tres elementos. A pesar de esto, tanto la información como el espacio de memoria ocupado será el mismo que en el caso de la representación en forma de matriz.

En este caso, con el fin de asignar una recaudación de 84 el viernes en la sala 2, lo haríamos de la siguiente forma:

```
recaudacionCine [5][2] := 84;
```

Si no nos queremos quedar con sólo dos valores del índice, de manera análoga a como lo hemos hecho al introducir las tablas de *n* dimensiones, ahora tendríamos:

```
var
   nombre: tabla [tamaño1] de tabla [tamaño2] ...
           de tabla [tamaño n] de tipoBase;
fvar
```

2.4. Acceso secuencial a una tabla

Una vez sabemos cómo declarar una tabla y cómo referenciar sus elementos con el objetivo de poder operar con éstos, vamos a introducir, ahora, algoritmos para procesar una tabla sobre la base de una acceso secuencial.

Ya sabemos...

... que la filosofía de acceso secuencial se basa en acceder a los elementos de manera ordenada: antes de considerar el elemento *i*-ésimo tenemos que haber considerado los *i* –1 anteriores.

El objetivo de este apartado es el de ver cómo, después de aplicar el modelo de secuencia en las tablas, se pueden adaptar los esquemas de recorrido y búsqueda que ya conocemos para aplicarlos a una tabla.

2.4.1. Esquema de recorrido aplicado a tablas

Hacer un recorrido sobre una tabla consistirá en un procedimiento que efectúa sobre cada elemento de la tabla una determinada operación de tratamiento.

Éste es un planteamiento muy general, cuyo punto importante, sin embargo, es que tenemos una operación que debemos aplicar a todos los elementos de la tabla. Por otro lado, ya nos hemos encontrado con una situación similar al hablar de secuencias, ¿por qué no intentar, entonces, aprovechar lo que ya tenemos?

En términos de secuencias, un planteamiento como el anterior obedecía a una aplicación del esquema del recorrido. Se tratará, por lo tanto, de ver que nuestra tabla es, de hecho, una secuencia de datos y que el esquema de recorrido se puede adaptar de la manera correspondiente. 

Con el fin de aplicar el modelo de secuencia a una tabla tenemos que, básicamente y de una manera informal, establecer el elemento inicial, el elemento final y un mecanismo de acceso que nos permita ir adelantando desde el elemento inicial hacia el elemento final.

– Elemento inicial: tenemos que saber dónde empieza el recorrido. Si tenemos que recorrer toda la tabla, el índice tendrá que empezar referenciando el primer elemento de la tabla. Éste es el que ocupa la primera posición.

– Elemento final: tenemos que saber dónde acaba el recorrido. Éste nos vendrá dado por el último valor que puede tomar el índice: la dimensión de la tabla. Si la tabla tiene capacidad para *tamaño* elementos, el índice no podrá exceder este valor, ya que estaríamos haciendo referencia a un elemento inexistente. Por otro lado, una situación en que el índice hace referencia al último valor de la tabla, corresponde a una tabla totalmente llena, pero la situación no siempre será esta. No obstante, de momento, supondremos que éste es el caso, y más adelante ya veremos en qué afecta la consideración de tabla parcialmente llena.

Es necesario distinguir...

... entre el hecho de que una tabla tenga capacidad para un cierto número de elementos y que a todos estos elementos se les haya asignado un valor.

– Mecanismo de acceso: necesitamos alguna forma de avanzar en el recorrido. Así, a través del índice, que tiene que ser un entero, podemos acceder al siguiente elemento de la tabla sencillamente incrementando en 1 su valor.

Una situación típica de actualización...

... si la variable es *i*, haríamos

i := i + 1

Recordemos ahora el esquema de recorrido que ya conocemos y cómo, a partir de las consideraciones hechas, lo aplicamos a una tabla.

```
prepararSecuencia
inicioTratamiento
mientras no finSecuencia hacer
        tratarElemento
        avanzarSecuencia
fmientras
tratamientoFinal
```

1) *prepararSecuencia*: en esta operación procedemos a que la variable que utilizamos como índice haga referencia al primer elemento de la tabla. En otras palabras, inicializamos el índice a 1.

2) *inicioTratamiento*: donde haremos todas las operaciones previas al recorrido propiamente dicho. Por ejemplo, la operación de llenar la tabla si es necesario.

3) *finSecuencia:* tendremos que verificar si la variable índice ya ha llegado a su valor final.

4) *tratarElemento*: aquí trataremos el elemento referenciado actualmente por la variable índice.

5) *avanzarSecuencia*: consistirá en la actualización de la variable índice incrementando en 1 su valor.

6) *tratamientoFinal*: lo tendremos o no dependiendo de si el tratamiento que hay que realizar en la tabla así lo requiere.

Un ejemplo nos ayudará a verlo un poco más claro.

Ejemplo

Supongamos que disponemos, en el dispositivo de entrada, de una secuencia de enteros con las calificaciones de 50 alumnos de una asignatura. Nos piden que calculemos la nota media y el número de alumnos que tienen una nota inferior a esta media. El último elemento de la secuencia es un –1.

En primer lugar, tenemos que especificar el algoritmo:

```
Promedio: real
InfPromedio: entero
{ Pre: Por el canal de entrada leemos una secuencia de 50 enteros acabada en –1 }
Calificaciones
{ Post: Promedio contiene el promedio de los 50 elementos de la secuencia e InfPromedio,
el número de elementos de la secuencia con un valor inferior a Promedio }
```

Una posible aproximación a la solución podría ser la siguiente. Dado que sabemos que hay 50 calificaciones, las podemos leer del dispositivo de entrada y cargarlas en una tabla, calculando la nota media y, así, poder ver después cuántas de estas calificaciones son inferiores a la media.

```
algoritmo Calificaciones
    const
        M: entero = 50;
    fconst
    var
        t: tabla [M] de entero;
        n, i, suma, infPromedio: entero;
        Promedio: real;
    fvar
    n := leerEntero();
    i := 1;
    Suma := 0;
    mientras no (n = –1) hacer
        t[i] := n;
        suma := suma + t[i];
        i := i + 1;
        n := leerEntero()
    fmientras
```

```
    Promedio := enteroAReal(suma) / enteroAReal (M);
    infPromedio := 0;
    i := 1;
    para i := 1 hasta M hacer
        si enteroAReal(t[i]) < promedio entonces
            infPromedio := infPromedio +1
        fsi
    fpara
falgoritmo
```

¿Qué sucedería si no supiésemos que el número de elementos es exactamente 50, sino como máximo 50? este hecho afectaría claramente en la manera de plantear el recorrido. Tendremos que pasar a trabajar con una tabla con capacidad para un máximo de 50 elementos, a pesar de que posiblemente no se llene en su totalidad.

Que una tabla...

... no se llene en su totalidad quiere decir que no todas las posiciones de la tabla contendrán un valor significativo que tratar.

En estos casos, es necesario que establezcamos algún sistema para especificar qué parte de la tabla utilizamos y, por tanto, cuál es el último elemento. Ciertamente, la determinación de cuál es el último elemento nos afectará en la condición de final de recorrido. Tenemos dos posibilidades para tratar esta situación:

La posición del último elemento...

... no tiene por qué coincidir con la determinada por el último valor del índice.

1) Utilizar una variable entera que nos indique hasta qué posición tenemos llena la tabla. En este caso, tendríamos que tener cuidado de que, al recorrer la tabla, la variable índice no supere nunca el contenido de esta variable entera.

2) Utilizar una marca o centinela: guardar un valor no significativo después del último valor para marcar el final de la parte utilizada de la tabla. Notemos que, en este caso, en el que todavía se usa una variable menos que en la anterior, se tiene que declarar la tabla con capacidad para un elemento más y, así, poder contemplar el caso en que la tabla esté llena hasta el final.

Veamos cómo se tendría que reescribir el ejemplo anterior considerando que de la secuencia de entrada sólo sabemos que tiene como máximo 50 calificaciones. Lo haremos para cada una de las dos posibilidades anteriormente comentadas con respecto a cómo tratar el recorrido de una tabla parcialmente llena.

Ejemplo

Consideremos el mismo problema del ejemplo anterior pero recordando que tenemos como máximo 50 calificaciones y utilizando un contador de las posiciones con elementos significativos de la tabla. Supondremos que hay, como mínimo, una calificación.

En este caso, la especificación quedaría adaptada como:

```
Promedio: real
InfPromedio: entero
{ Pre: por el canal de entrada leemos una secuencia no vacía de como mucho 50 enteros
acabada en –1 }
Calificaciones
{ Post: promedio contiene el promedio de los elementos de la secuencia e InfPromedio, el
número de elementos de la secuencia con un valor inferior a promedio }
```

```
algoritmo calificaciones
    const
        M: entero = 50;
    fconst
    var
        t : tabla [M] de entero;
        cuantos : entero;{ Indicará cuántas posiciones significativas tenemos en la tabla }
        n, i: entero;
        suma, infPromedio: entero
        promedio : real
    fvar
    n := leerEntero();
    cuantos := 0;
    suma := 0;
    mientras no n = –1 hacer
        cuantos := cuantos + 1;
        t[cuantos] := n;
        suma := suma + t[cuantos];
        n := leerEntero()
    fmientras
    promedio := enteroAReal(suma) / enteroAReal (cuantos);
    InfPromedio := 0;
    para i := 1 hasta cuantos hacer
        si enteroAReal(t[i]) < Promedio entonces
            infPromedio := infPromedio + 1
        fsi
    fpara
falgoritmo
```

Ejemplo

Consideremos el mismo problema de las 50 calificaciones como máximo pero utilizando, ahora, un centinela para marcar la primera posición vacía de la tabla.

```
algoritmo calificaciones
    const
        M: entero = 50;
        centinela: entero = –1 { Utilizaremos como centinela la misma marca que nos señala
                                 el final de secuencia }
    fconst

    var
        t: tabla [M + 1] de entero;
        n, i, suma, infPromedio: entero;
        Promedio: real;
    fvar

    n := leerEntero();
    i := 1;
    suma := 0;
    mientras no n = –1 hacer
        t[i] := n;
        suma := suma + t[i];
        i := i + 1;
        n := leerEntero()
    fmientras
    { A continuación del último elemento, ponemos la marca de final de
    valores significativos }
    t[i] := centinela;
    promedio := enteroAReal(suma) / enteroAReal(i – 1);
    infPromedio := 0;
    i := 1;
    mientras t[i] ≠ centinela hacer
        si enteroAReal (t[i]) < promedio entonces
            infPromedio := InfPromedio + 1
        fsi
        i := i + 1
    fmientras
falgoritmo
```

Tenemos que declarar...

… la tabla de un elemento más, ya que la secuencia puede contener hasta 50 calificaciones más otra para el centinela.

Fijémonos en que...

… ahora la condición de final de recorrido de la tabla la tenemos que basar en la variable que nos dice en qué posición tenemos el último elemento significativo.

Resumiendo, podemos decir que la condición de seguimiento del recorrido puede adoptar tres formas diferentes:

1) Comparación de la variable que actúa como índice con el valor más alto posible de éste de acuerdo con la declaración de la tabla.

2) Comparación con la variable entera que nos marca hasta qué posición está llena la tabla.

3) Comparación del elemento guardado en la posición referenciada por el índice con el valor que actúa de centinela.

2.4.2. Esquema de búsqueda aplicado a tablas

El objetivo del esquema de búsqueda sobre un modelo de secuencia es buscar el primer elemento que cumpla una determinada condición, teniendo en cuenta, sin embargo, que este elemento puede no existir y que la búsqueda la hacemos a partir de un acceso secuencial a los datos. Ésta será, también, la idea con la que nos planteemos aplicarlo a una tabla.

Es importante remarcar que...

… la búsqueda se hace basándonos en un acceso secuencial, ya que, como veremos más adelante, el acceso directo a los elementos de una tabla nos ofrece otras posibilidades, a veces mucho más eficientes, para llevar a cabo una búsqueda.

Sabiendo aplicar el modelo de secuencia en las tablas y cómo plantear un acceso secuencial a los elementos de una tabla, tenemos todo lo necesario para poder aplicar el esquema de búsqueda a una tabla. Recordemos, igual que hemos hecho con el recorrido, cuál es el esquema de búsqueda y qué consideraciones debemos tener en cuenta para poder aplicarlo en una tabla.

```
prepararSecuencia
encontrado := falso;
inicioTratamiento
mientras no finSecuencia y no encontrado hacer
        actualizarEncontrado
        si no encontrado entonces
            tratarElemento
            avanzarSecuencia
        fsi
fmientras
tratamientoFinal
```

1) *prepararSecuencia*: en esta operación la variable que utilizaremos como índice referencia al primer elemento de la tabla. En otras palabras, inicializamos el índice a 1.

2) *inicioTratamiento*: haremos todas las operaciones previas a la búsqueda propiamente dicha; por ejemplo, la operación de llenar la tabla si es necesario.

3) *finSecuencia:* se verifica si la variable índice ya ha llegado al valor final.

4) *actualizarEncontrado*: se verifica si el elemento referenciado actualmente por la variable índice cumple o no la condición de búsqueda.

5) *tratarElemento*: trataremos el elemento referenciado actualmente por la variable índice.

6) *avanzarSecuencia*: se actualiza la variable índice incrementando en 1 su valor.

7) *tratamientoFinal*: lo tendremos o no, dependiendo de cuál sea el objetivo de la búsqueda.

Ahora podemos pasar a ver con un ejemplo cómo sería esta aplicación. A lo que prestaremos especial atención, en todo caso, es a la condición de finalización de la búsqueda. Efectivamente, recordemos que, al aplicar un esquema de búsqueda, ésta puede acabar por dos motivos:

1) Porque hemos encontrado el elemento buscado.
2) Porque hemos llegado al final de la secuencia (tabla en este caso).

La manera como expresaremos estas dos posibilidades se debe tener presente, después del bucle, cuando miremos si hemos encontrado o no, el elemento buscado.

Ejemplo

Supongamos que disponemos de una tabla de 100 elementos en la que tenemos que adivinar si entre estos 100 hay uno determinado.

La especificación de este algoritmo sería la siguiente:

```
t: tabla [100] de entero
encontrado: booleano
dado: entero
{ Pre: t = T es una tabla con 100 valores enteros y dado = DADO }
búsqueda
{ Post: encontrado es cierto en caso de que en la tabla T haya una posición que contiene el
valor DADO, de lo contrario tendríamos encontrado = falso }

algoritmo búsqueda
    const
        N: entero = 100;
    fconst
    var
        t: tabla [N] de entero;
        i, dado: entero;
        encontrado: booleano;
    fvar
    llenarTabla;
    dado := leerEntero();
```

Recordad que la acción *llenarTabla* ya la diseñamos en el apartado 2.2.2.

```
    i := 1;
    encontrado := falso;
    mientras i ≤ N y no encontrado hacer
        si t[i] = dado entonces
            encontrado := cierto
        sino
            i := i + 1
        fsi
    fmientras
    si encontrado entonces
        escribirCaracter('S')
    sino
        escribirCaracter('N')
    fsi
falgoritmo
```

Analicemos esta búsqueda. Fijémonos en que cuando salimos del **mientras** tenemos que distinguir entre las dos posibilidades ya comentadas de elemento encontrado o no encontrado. Veamos qué sucede en cada caso:

Estaría bien...

… que verificaseis que, efectivamente, ésta es la situación aunque el elemento se encuentre en la última posición.

1) *encontrado* = **cierto**. En este caso, la variable índice hace referencia a la posición de la tabla donde hemos encontrado el elemento buscado.

2) *encontrado* = **falso**. En este caso, el elemento buscado no se encuentra en la tabla y, por tanto, habremos salido del bucle **mientras** debido a que ya se ha recorrido toda la tabla; no se verifica la condición $i \leq N$ y la variable índice referencia una posición más allá del tamaño declarado. Esta situación también puede resultar indeseable debido a que se puede dar el caso de que el elemento siguiente al último valor del índice no esté definido. Esta situación comporta que al finalizar una búsqueda siempre deberemos tener presente el valor de *encontrado* antes de plantear cualquier operación que implique el índice de esa tabla.

¿Pensad qué...

... obtendríamos si igualásemos, por ejemplo, la variable índice a:

i := i + 1

cuando *i* es igual al ENTERO-MAX, ¿a qué elemento de la tabla se hace referencia?

Ejemplo

Por claridad...

... escribimos sólo la parte del algoritmo correspondiente a la búsqueda propiamente dicha.

Imaginemos que declaramos una tabla para reunir la recaudación de un cine durante una semana. De hecho, ya habíamos visto una:

```
var
    recaudacionCine: tabla [7] de entero;
fvar
…
dia := 1;
encontrado := falso;
mientras dia ≤ 7 y no encontrado hacer
    si recaudacionCine[dia] < 10 entonces
        encontrado := cierto
    fsi
    si no
        dia := dia + 1
    fsi
fmientras
…
```

Si queremos saber si hay un día de la semana en que la recaudación ha sido inferior a 10 aplicaremos el esquema de búsqueda tal y como hemos hecho antes.

En caso de que no haya ningún día con una recaudación inferior a 10, llegaremos al momento en que:

- *dia* = 7,
- *recaudacionCine*[7] ≥ 10 y, por tanto, continuaremos con *encontrado* = **falso**,
- *encontrado* = **falso** y por tanto haremos *dia* := dia + 1 y obtendremos *dia* = 8.

Fijémonos en que la variable índice, *dia*, nos quedará referenciando a una supuesta posición 8 dentro de la tabla *recaudacionCine*. Evidentemente, esta posición no existe, de modo que cualquier operación que queramos hacer con la misma inducirá a error.

Pero fijémonos en que, este problema, que la variable índice haga referencia a una posición inexistente, ya planteado en el esquema de búsqueda, también se puede encontrar al aplicar el esquema de recorrido. No obstante, en este caso la problemática que genera no es tan evidente, ya que en contadas ocasiones, al hacer un recorrido, tendremos que tratar de una manera especial el elemento referenciado al acabar la iteración. Esta situación es precisamente la que tenemos en el esquema de búsqueda.

De la misma forma: las situaciones comentadas en la aplicación de un recorrido a una tabla, en tanto que consideramos la posibilidad de no tener toda la tabla llena con valores significativos, son también perfectamente aplicables al caso de una búsqueda. Así, si tenemos que buscar un elemento en una tabla parcialmente llena, deberemos tener en cuenta las mismas consideraciones hechas acerca de la condición de final de secuencia, de forma que ahora esta condición incluirá, ya sea la comparación del índice con una variable que nos indica la última posición donde tenemos un valor significativo, ya sea la comparación del elemento referenciado actualmente por el índice con una marca o centinela. En este último caso, por ejemplo, la condición de finalización de la búsqueda del ejemplo anterior tomaría la siguiente forma:

```
...
mientras (t[i] ≠ centinela) y no encontrado hacer
...
```

2.4.3. Ejemplos de los esquemas de recorrido y búsqueda

En los ejemplos que siguen, supondremos que trabajamos con la tabla siguiente:

```
const
    N: entero = 100;
fconst
var
    t: tabla [N] de entero;
fvar
```

Ejemplo

Dada la tabla *t*, hay que obtener en qué posición se encuentra el elemento menor.

t: **tabla**[N] **de entero**
posPeque: **entero**
{ Pre: *t* = *T* contiene *N* valores enteros }
posicionMenor
{ Post: *posPeque* indica la posición de *t* que contiene el elemento menor, es decir t[*posPeque*] ≤ t[*i*] para *i* desde 1 hasta *N* }

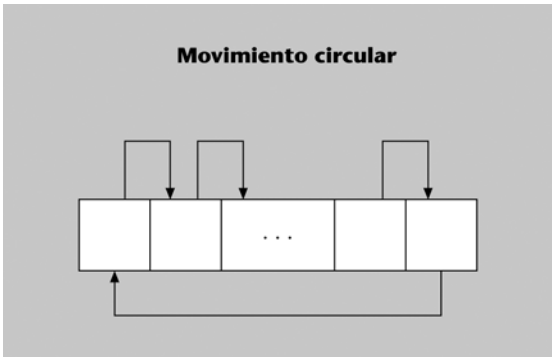
En este caso, está claro que se trata de hacer un recorrido de todos los elementos de la tabla, habiendo ido guardando la posición que ocupa el elemento menor encontrado hasta el momento.

```
...
posPeque := 1;
i := 2;
mientras i ≤ N hacer
    si t[i] < t[PosPeque] entonces
            PosPeque := i
    fsi
    i := i + 1
fmientras
escribirEntero(PosPeque);
...
```

Observad que empezamos con i := 2, de manera que la primera pregunta es si el segundo elemento es menor que el primero.

Ejemplo

Hay que mover los valores de una tabla de tamaño $N > 0$ de acuerdo con una rotación circular tal y como se muestra en el gráfico siguiente:



La especificación de este problema sería:

t: **tabla**[N] **de entero**
{ Pre: *t* = *T* contiene *N* valores enteros }
rotacionCircular
{ Post: *t*[1] = *T*[*N*] y para *i* desde 2 hasta *N* *t*[*i*] = *T*[*i*–1] }

Una posible solución es la siguiente:

Fijémonos en que...

… el uso de esta variable auxiliar es imprescindible, igual que lo es para intercambiar el conte nido de dos posiciones.

- Primero guardamos el contenido de la última posición en una variable auxiliar.
- Después efectuamos un recorrido hacia atrás del vector (desde la posición *N* hasta la 2) copiando sobre cada elemento el elemento anterior.
- Finalmente guardamos el valor de la variable auxiliar en la primera posición.

Este recorrido hacia atrás...

… lo podemos aplicar a una tabla gracias al acceso directo y a que conocemos su longitud. Por tanto, nos podemos situar directamente en el último elemento.

El recorrido lo hacemos en orden decreciente con el fin de no "pisar" los valores del vector. Tenemos que haber copiado el *t*[9] a *t*[10] antes de haber movido *t*[8] a *t*[9], ya que, de lo contrario, perderíamos el contenido de *t*[9].

Ejemplo

Hay que decidir si todos los elementos de una tabla de tamaño $N > 0$ son superiores a un valor dado.

Como siempre, primero especificamos el problema:

```
t: tabla[N] de entero
dado: entero
todosSuperiores: booleano
{ Pre: t = T contiene N valores enteros y dado = DADO }
decidirSiSuperiores
{ Post: todosSuperiores es cierto si para i desde 1 hasta a N t[i] ≥ DADO, contrariamente todos-
Superiores es falso }
```

Podemos resolver esta situación mediante un esquema de búsqueda en el que la condición de búsqueda sería que el elemento fuese inferior a uno dado.

```
…
dado := leerEntero( );
i := 1;
encontrado := falso;
mientras (i < N) y no encontrado hacer
    si t[i] < dado entonces
        encontrado := cierto
    sino
        i := i + 1
    fsi
fmientras
si no encontrado entonces
    escribirCaracter('S')
sino
    escribirCaracter('N')
fsi
…
```

Ejemplo:

Se quiere invertir el contenido de un vector de tamaño $N > 0$.

Veamos cómo especificamos este problema:

t: **tabla**[N] **de entero**
{ Pre: *t* = *T* contiene *N* valores enteros }
invertirVector
{ Post: para *i* desde 1 hasta *N* *t*[*i*] = *T*[*N* – i + 1] }

Una estrategia podría ser la siguiente:

- Intercambiar el primer elemento con el último.
- Intercambiar el segundo elemento con el penúltimo.
- Y así sucesivamente hasta llegar al elemento central.

Se debe llevar a cabo, por tanto, un recorrido de la primera mitad del vector, siendo el tratamiento el intercambio con el elemento correspondiente de la segunda mitad.

```
...
i := 1;
mientras i ≤ N div 2 hacer
    aux := t[i];
    t[i] := t[N – i + 1];
    t[N – i + 1] := aux;
    i := i + 1
fmientras
...
```

Respecto al límite del recorrido, aclaramos que:

1) Si el número de elementos es par, (por ejemplo $N = 10$) entonces el límite del recorrido será 10 **div** 2 = 5, siendo el último intercambio el que tiene lugar entre las posiciones 5 y 6 (10 – 5 + 1).

2) Si el número de elementos es impar, entonces tenemos un elemento, el elemento central, que no se tiene que intercambiar. Si tenemos, por ejemplo, $N = 11$, entonces el último intercambio es el que tiene lugar entre las posiciones 5 y 7, la posición 6 no se debe intercambiar. Esta situación también queda automáticamente contemplada con el *N* **div** 2 como límite del recorrido, ya que 11 **div** 2 = 5.

Ejemplo

Un vector es capicúa si en la primera y última posición tenemos el mismo valor, la segunda y penúltima también tienen el mismo, y así sucesivamente hasta al centro del vector. Se trata de construir un algoritmo que verifique si un vector es capicúa.

Lo podemos especificar de la siguiente manera:

t: **tabla**[N] **de entero**
esCapicua: **booleano**
{ Pre: *t* = *T* contiene *N* valores enteros }
decidirSiCapicua
{ Post: *esCapicua* es **cierto** si se verifica *T*[*i*] = *T*[*N* – *i* + 1] para *i* desde 1 hasta *N* **div** 2. Contrariamente, *esCapicua* será **falso** }

Así pues, para comprobar si un vector es capicúa tenemos que ir comparando los elementos tomándolos por pares, al igual que hacemos al invertir un vector. Ahora, sin embargo, la comparación se acaba al encontrar algún par que no coincida. Se trata, por tanto, de un esquema de búsqueda. 

```
encontrado := falso;
i := 1;
mientras (i ≤ N div 2) y no encontrado hacer
    si (t[i] ≠ t[N – i + 1]) entonces
        encontrado := cierto
    fsi
    si no encontrado entonces
        i := i + 1
    fsi
fmientras
esCapicua := no encontrado;
…
```

Ejemplo

Escribir un algoritmo para multiplicar dos matrices.

Recordemos primero que, para multiplicar dos matrices, se deben verificar las condiciones siguientes:

- La primera matriz debe tener tantas columnas como filas la segunda.
- La matriz resultado debe tener tantas filas como la primera y tantas columnas como la segunda.

Así, si multiplicamos una matriz A, de N filas y M columnas, y una matriz B de M filas y P columnas, los elementos de la matriz C resultado se calcularán según:

$$C_{ij} = \sum_{k=1}^{M} a_{ik} b_{kj} \text{ con } i = 1, \ldots, N, j = 1, \ldots, P$$

La especificación se basará, en este caso, en la aplicación de esta fórmula.

```
a: tabla [ N, M] de entero
b: tabla [ M, P] de entero
c: tabla [ N, P] de entero
{ Pre a = A: es una matriz de enteros con N filas y M columnas y b = B es una matriz de enteros
de M filas y P columnas }
productoMatrices
{ Post: para i desde 1 hasta N y j desde 1 hasta P, todo elemento c[i,j] es la suma de los productos A[i,k] * B[k,j] para k desde 1 hasta M }
```

Por tanto, el procedimiento consiste en los pasos siguientes:

1) Recorrer las N filas de la tabla a.

2) Para cada fila, recorrer todas las P columnas de la matriz b.

3) Para cada columna, hacer la operación de la expresión anterior para calcular el elemento correspondiente de la matriz resultante.

```
algoritmo productoMatrices
    const
        N: entero = 6;
        M: entero = 8;
```

```
        P: entero = 7;
    fconst
    var
        a: tabla [ N, M] de entero;
        b: tabla [ M, P] de entero;
        c: tabla [ N, P] de entero;
        i, j, k: entero;
        v: entero;
    fvar
    llenarTabla(a);
    llenarTabla(b);
    para i := 1 hasta N hacer
        para j := 1 hasta P hacer
            v := 0;
            para k := 1 hasta M hacer
                v := v + a[i, k] * b[k, j]
            fpara
            c[i, j] := v
        fpara
    fpara
falgoritmo
```

Sería interesante que...

... en este algoritmo identificaseis la anidación de recorridos que existen; es decir, un recorrido dentro de otro.

2.5. Acceso directo a una tabla

Una de las características más importantes de las tablas es la posibilidad de acceder directamente a cualquier elemento de la tabla, dada la posición que éste ocupa. Hasta ahora, sin embargo, hemos explotado poco esta posibilidad de acceso directo. Básicamente nos hemos limitado a recorrer los elementos de la tabla siguiendo un orden secuencial de los valores del índice.

En este apartado veremos cómo sacar partido a esta característica presentando procedimientos para ordenar los elementos de una tabla y para buscar un elemento de una forma más eficiente que con el esquema de búsqueda.

2.5.1. Importancia de los datos ordenados. Búsqueda dicotómica

Para buscar si un determinado elemento está presente o no en una tabla, hemos aplicado el esquema usual de búsqueda, en el que se empieza por el inicio de la tabla y se van examinando los elementos uno a uno. Este esquema, denominado también **búsqueda lineal**, es el que, de hecho, podemos aplicar siempre. No obstante, en el caso de que los elementos de la tabla estén ordenados, se puede aplicar un esquema diferente que saca partido de la posibilidad de acceso directo a los elementos de la tabla. Este esquema es la búsqueda dicotómica.

El algoritmo de búsqueda dicotómica se inspira en una idea sencilla que con seguridad habréis aplicado en alguna ocasión. Suponed que tenéis que buscar el teléfono de un amigo en la guía telefónica. ¿Verdad que no empezáis por la primera página y leéis todos los nombres hasta que encontréis el de vuestro amigo? A menudo, lo que hacemos es tomar una página cualquiera de la guía, al azar, y comparar la primera letra del apellido de nuestro amigo con la primera letra del nombre que aparece en la página que hemos seleccionado al azar. A partir de aquí, determinamos en qué parte de la guía, anterior o posterior al punto donde nos encontramos, se puede encontrar el nombre que buscamos, y continuamos con

el mismo procedimiento hasta llegar a la página donde se encuentra. Es más, podemos afirmar si el nombre se encuentra o no en la guía sin necesidad de leer todos los nombres gracias a que sabemos que en la guía los nombres están dispuestos según un orden alfabético.

Vamos a resumir esta idea pero aplicada a una tabla, lo que nos dará las bases para escribir el algoritmo correspondiente.

1) Tomamos el elemento del medio de la tabla y lo comparamos con el elemento buscado:

- Si el elemento del medio es menor que el elemento buscado, entonces descartaremos la primera mitad de la tabla (ya que sólo podrá contener valores menores que el buscado).
- Si, en cambio, el elemento del medio es mayor que el buscado, descartamos la mitad posterior de la tabla.

2) Aplicamos el mismo esquema en la mitad de tabla no descartada como si se tratase de una tabla entera.

Estos dos pasos se repetirán hasta quedarnos con una parte no descartada de la tabla de sólo un elemento.

El siguiente algoritmo aplica esta idea:

```
algoritmo busquedaDicotomica
    const N: entero = 100;
    fconst
    var
        t: tabla [N] de entero;
        dado: entero;
        inicio, fin, medio: entero;
    fvar
    llenarTabla(t);
    dado := leerEntero();
    inicio := 1;
    fin := N;
    mientras inicio ≠ fin hacer
        medio := (inicio + fin) div 2;
        si t[medio] < dado entonces
            inicio := medio + 1
            sino fin := medio
        fsi
    fmientras
    si t[inicio] = dado entonces
        escribirCaracter('S')
        sino escribirCaracter('N')
    fsi
falgoritmo
```

Es importante fijarse en que al actualizar los límites de la tabla no descartada, en el caso de que no se cumpla la condición $t[medio] < dado$ (y, por tanto, lo que se cumplirá será $t[medio] \geq dado$), es necesario que hagamos fin := medio y no fin := medio − 1, ya que si no estaríamos considerando sólo la posibilidad $t[medio] > dado$ y excluiremos el posible caso $t[medio] = dado$.

Vemos claramente que el algoritmo de búsqueda dicotómica es bastante más eficiente que el de búsqueda lineal; ya que mientras que la búsqueda lineal descarta un elemento en cada iteración, la búsqueda dicotómica descarta cada vez la mitad de los que quedan. Debemos tener presente que únicamente se puede aplicar si sabemos con seguridad que el vector está ordenado.

2.5.2. Ordenación por selección

La ordenación de los elementos de una tabla es un tema bastante complejo, ya que existen métodos muy distintos de ordenación, con variantes dentro de cada método, algunos bastante sofisticados, que utilizan técnicas de programación más avanzadas y que están fuera del alcance de este módulo.

Ya hemos visto, por otro lado, que el hecho de disponer de una tabla ordenada puede incrementar la eficiencia en el momento de realizar una búsqueda sobre la misma.

A pesar de que no entraremos en una exposición y comparación de métodos de ordenación, sí consideraremos uno para presentar las ideas en las que se basan algunos de estos algoritmos. El algoritmo que veremos es el llamado de **selección directa**.

Otros métodos de ordenación...

… junto con consideraciones sobre su complejidad y eficiencia se pueden encontrar en la bibliografía recomendada.

El procedimiento que sigue este método para ordenar los elementos de una tabla de pequeño a grande se basa en la idea de dividir la tabla que se está ordenando en dos partes:

1) Una primera parte ordenada, en la que todos los elementos que la forman están ordenados.

2) Una segunda parte formada por todos los elementos que todavía están por ordenar.

El algoritmo de ordenación por selección directa incluye los pasos siguientes:

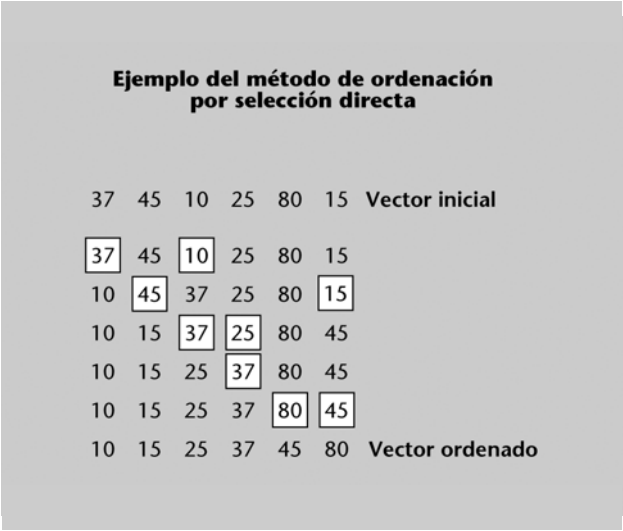
- Seleccionar el menor elemento e intercambiarlo con el que ocupa la primera posición (con esto conseguimos una primera parte ordenada de un solo elemento).

- Tomar el menor de los restantes (posiciones de la 2 hasta la *N*) e intercambiarlo con el que ocupa la segunda posición.

- Repetir sucesivamente este procedimiento hasta que sólo quede la *N*-ésima posición, la cual contendrá, necesariamente, el elemento mayor.

Ejemplo:

Veamos un ejemplo de la secuencia de intercambios que tendrían lugar al ordenar un vector aplicando este método:



Para escribir un algoritmo que nos implemente este método, fijémonos en que las operaciones que constan se reducen a un recorrido en el que, en cada iteración, debemos llevar a cabo una búsqueda del elemento menor (búsqueda que realizaremos con su correspondiente recorrido). La idea del algoritmo sería, pues:

```
algoritmo seleccionDirecta
    ...
    i := 1;
    { preparar }
    mientras i < N hacer
        { tratar: buscar el elemento menor de t[i],...,t[N] y
        intercambiarlo con t[i] }
        i := i + 1;
    { avanzar }
    fmientras
    ...
falgoritmo
```

Fijémonos en que...

... el recorrido se puede hacer con $i < N$ en lugar de $i \leq N$, ya que el último elemento ya quedará ordenado de forma automática.

De manera que, desarrollando las operaciones de búsqueda del elemento menor y de intercambio, el algoritmo nos quedaría como:

```
algoritmo seleccionDirecta
    const
    N: entero = 100;
    fconst
    var
        t: tabla [N] de entero;
        i, j, posMin: entero;
        aux: entero;
    fvar
    llenarTabla(t);
    i := 1;
    mientras i < N hacer
        posMin := i;
        j := i + 1;
        mientras j ≤ N hacer
            si t[j] < t[posMin] entonces
                posMin := j
            fsi
            j := j + 1
        fmientras
        aux := t[i];
        t[i] := t[posMin];
        t[posMin] := aux;
        i := i + 1
    fmientras
falgoritmo
```

3. Tuplas

Una tupla es un objeto que contiene otros objetos denominados *campos*. Cada uno de estos subobjetos tendrá un nombre para identificarlo dentro de la tupla, así como un tipo de datos.

Podemos considerar...

... los elementos de una tupla como propiedades del objeto.

3.1. Declaración

La declaración de una tupla consiste en definir cuáles serán los elementos o campos que contendrá, indicando el nombre y el tipo de datos de cada uno. Esta acción, en lenguaje algorítmico, se lleva a cabo dentro de un apartado acotado por las palabras clave **tupla** … **ftupla** según la siguiente sintaxis:

```
var
    nombre: tupla
        nombreCampo1: tipo1;
        nombreCampo2: tipo2;
        . . .
            ftupla
fvar
```

En esta declaración se indican los campos que tendrá la tupla, sus tipos y el nombre con que agrupamos estos campos:

1) *nombre*: identificador de la variable de tipo tupla que estamos construyendo. Con este único identificador quedarán agrupados todos los campos que definimos después. Éste será, por lo tanto, el identificador que tendremos que utilizar después, junto con el identificador del campo correspondiente, para referirnos a cada uno de los campos.

2) *Campoi*: es el identificador que definirá el campo *i*. Una tupla estará formada por varios campos, y cada uno tendrá un identificador, único dentro de la tupla, que lo diferenciará del resto de los campos. Mediante este identificador y el identificador de la tupla nos referiremos al valor del campo.

3) *tipoBasei*: tipo del campo *i*. Indica qué tipo de valores podrá contener este campo.

Veamos a continuación algunos ejemplos de declaración de tuplas:

Ejemplo

```
var
    proveedor1, proveedor2: tupla
            nombre: tabla [15] de caracter;
```

```
                    telefono, fax: entero;
                        ftupla
    fvar
```

Tanto la variable *proveedor*1 como *proveedor*2, así definidas, son tuplas. Los campos de una tupla no tienen sentido por sí solos, es decir, no son variables independientes. El campo *nombre* no puede ser tratado como una variable independiente, ya que sólo tiene sentido dentro de la variable *proveedor*1 o de la variable *proveedor*2. O sea, el campo *nombre* sólo tiene sentido si lo vemos como un nombre del proveedor.

El orden de declaración de los campos dentro de una tupla es indiferente.

También podríamos haber definido un nuevo tipo de datos utilizando el constructor de tipos:

Observad...

... el constructor de tipos en el módulo didáctico "Introducción a la algorítmica".

```
tipo
    proveedor = tupla
                    nombre: tabla [15] de caracter;
                    telefono, fax: entero;
                ftupla
ftipo
```

Con el tipo *proveedor* así definido, podemos definir las dos variables *proveedor*1 y *proveedor*2:

```
var
        proveedor1, proveedor2: proveedor;
fvar
```

Veamos, a continuación, algunos ejemplos de declaración de tuplas:

Ejemplo 1

```
var
    coordenadas: tupla
                    x, y: real;
                 ftupla
fvar
```

Ejemplo 2

```
tipo
    listaMes: { enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre,
              noviembre, diciembre };
    fecha = tupla
                dia: entero;
                mes: listaMes;
                año: entero;
            ftupla
ftipo
```

Ejemplo 3

Los campos...

… de una tupla también pueden ser tablas, al igual que los elementos de una tabla pueden ser tuplas.

```
tipo
    nota = (E, D, C, B, A);
    alumno = tupla
                codigoMatricula: entero;
                nombre: tabla [25] de caracter;
```

```
            direccionE: tabla [40] de caracter;
            notasPEC: tabla [5] de nota;
            calificacion: nota;
        ftupla
ftipo
```

3.2. Operaciones elementales de acceso

Ya sabemos cómo definir variables de tipo tupla en nuestros algoritmos. Como ya hemos visto, una variable de este tipo es, en realidad, un grupo de campos, y son estos campos los que contienen la información. Hemos visto, también, que estos campos no pueden ser tratados como objetos independientes, ya que forman parte de un objeto más general que los engloba y que les da sentido.

Por tanto, si queremos hacer referencia a cualquiera de los campos de una tupla, debemos conseguirlo indicando la tupla a la que pertenece.

En el lenguaje algorítmico, escribiremos las referencias a un campo de una tupla indicando el nombre de la variable de tipo tupla, seguida de un punto y del nombre del campo:

nombreTupla.nombreCampo

Evidentemente,...

... es necesario que el campo se haya definido como parte de la tupla.

Ejemplo

Con la siguiente declaración, ya vista antes:

```
tipo
    proveedor = tupla
                    nombre: tabla [15] de caracter;
                    telefono, fax: entero;
                ftupla
ftipo
var
    proveedor1, proveedor2: proveedor;
fvar
```

Las referencias siguientes serán correctas:

```
proveedor1.nombre[1]    { el primer carácter del campo nombre de la tupla proveedor1 }
proveedor2.telefono
proveedor1.fax
```

Las referencias siguientes no serán correctas:

```
nombre[1]    { no podemos hacer referencia al campo sin indicar el nombre de la tupla }
proveedor.telefono    { no podemos utilizar el tipo proveedor como si fuese una variable }
fax.proveedor1    { el orden tiene que ser nombreTupla.nombreCampo, y no al revés }
```

3.2.1. Asignación y consulta

Asignar valores a una variable de tipo tupla significa llenar los campos. Para hacerlo tenemos dos opciones: 

a) Asignar uno a uno los valores a cada campo de la tupla.

b) Asignar directamente a la tupla otra tupla del mismo tipo, o bien el resultado de una función que retorna una tupla del mismo tipo.

Siguiendo con el mismo ejemplo:

```
proveedor1.telefono := 939876544;
proveedor2.telefono := 934567891;
```

Como toda asignación,...

... el valor asignado debe ser del mismo tipo que el campo de la tupla a la que lo asignamos. En este caso, es un entero.

asigna el valor 939876544 al campo *teléfono* de la variable *proveedor*1 y el valor 934567891 al campo *teléfono* de la variable *proveedor*2.

Para **consultar** los valores contenidos en los campos de una tupla, debemos consultar los valores para cada campo por separado, y sólo es necesario escribir la referencia del campo.

Suponiendo las siguientes declaraciones:

```
tipo
    coordenadas = tupla
                      x, y: real;
                  ftupla
ftipo
var
    a, b, puntoMedio: coordenadas;
fvar
```

Podemos calcular el punto medio entre los puntos *a* y *b* haciendo:

```
puntoMedio.x := (a.x + b.x) / 2.0;
puntoMedio.y := (a.y + b.y) / 2.0;
```

3.2.2. Lectura y escritura por los canales de entrada/salida estándar

Evidentemente, los valores que asignemos a los campos de una tupla pueden provenir de un dispositivo de entrada (lectura), y también podemos enviar a un dispositivo de salida los campos de una tupla (escritura).

Ejemplos de escritura y lectura	
a.x := leerReal()	Lee un dato del dispositivo de entrada y lo asigna al campo x de la tupla a.
a := leerReal ()	No tiene sentido, ya que implica operar con la tupla entera (provocaría error).
escribirReal (a.x)	Escribe en el dispositivo de salida el valor contenido en el campo x de la tupla a.
escribirReal(a)	No tiene sentido, ya que implica operar con la tupla entera (provocaría error).

Así pues, si queremos llenar todos los campos de una tupla con valores provenientes de la entrada estándar o escribir el contenido de una a la salida estándar, podemos definir nuestras propias funciones auxiliares para hacerlo. Fijémonos en el siguiente ejemplo:

```
algoritmo puntoMedio
    tipo
        coordenadas = tupla
                          x, y: real;
                      ftupla
    ftipo
    var
        a, b: coordenadas;
    fvar
        { Pre: leeremos cuatro reales del canal estándar X1, Y1, X2, Y2, que representan las
          coordenadas de dos puntos, A y B respectivamente }
        a := obtenerPunto();
        b := obtenerPunto();
        escribirPunto(calculoPuntoMedio( a, b ));
        { Post: hemos escrito en el canal estándar las coordenadas del punto medio entre A y
          B, que son (X1 + X2)/2, (Y1 + Y2)/2 }
falgoritmo

funcion obtenerPunto (): coordenadas
    var
       p: coordenadas;
    fvar
    { Pre: leemos del canal estándar dos reales X e Y }
    p.x := leerReal();
    p.y := leerReal();
    retorna p
    { Post: p.x = X y p.y = Y }
ffuncion
accion escribirPunto (ent p: coordenadas)
    { Pre: p.x = X y p.y = Y }
    escribirReal(p.x);
    escribirCaracter(',');
    escribirReal(p.y);
    { Post: hemos escrito en el canal estándar "X,Y" }
faccion
funcion calculoPuntoMedio(p1, p2: coordenadas): coordenadas
    var
       p: coordenadas;
    fvar
    { Pre: p1.x = X1 y p1.y = Y1 y p2.x = X2 y p2.y = Y2 }
    p.x := (p1.x + p2.x) / 2;
    p.y := (p1.y + p2.y) / 2;
    retorna p
    { Post: p1 y p2 no han cambiado y p.x = (X1 + X2)/2 y p.y = (Y1 + Y2)/2 }
ffuncion
```

Fijaos en que...

... las funciones pueden retornar tuplas, y podemos asignar el retorno de una función directamente a una variable, siempre y cuando sea exactamente del mismo tipo.

Fijaos en los aspectos siguientes:

- Una función puede retornar una tupla (**funcion** obtenerPunto).
- Una acción puede recibir parámetros de tipo tupla (**accion** escribirPunto).
- Podemos asignar a una tupla el retorno de una función, si coinciden los tipos (a := obtenerPunto();).
- Podemos utilizar el retorno de una función como parámetro de una acción, si coinciden los tipos (escribirPunto(calculoPuntoMedio(a, b));).

3.3. Otros tipos de tuplas

3.3.1. Tuplas de tuplas

Los campos de una tupla pueden ser de cualquiera de los tipos de datos que hemos visto en el lenguaje algorítmico. Así pues, nos podemos encontrar con que alguno de los campos de una tupla sea, también, una tupla.

Ejemplo 1

Queremos definir un tipo de datos para manipular un periodo de tiempo. Podemos definir un periodo con la fecha de inicio y la fecha de finalización, de manera que podríamos hacer la siguiente definición en lenguaje algorítmico:

```
tipo
periodo = tupla
            diaInicio, diaFin: entero;
            mesInicio, mesFin: { enero, febrero, marzo, abril, mayo, junio, julio,
            agosto, septiembre, octubre, noviembre, diciembre };
            añoInicio, añoFin: entero;
          ftupla
ftipo
```

Así pues, agrupamos día, mes y año de inicio, y día, mes y año de finalización del período dentro de una tupla que llamamos *periodo*. Las tuplas permiten agrupar información relacionada de forma lógica, y esto es lo que estamos haciendo, agrupamos los seis campos que definen un período. A pesar de ello, si nos fijamos bien, todavía no estamos utilizando la tupla de forma óptima. Habíamos dicho que la agrupación de datos tenía sentido si nos permitía evitar la dispersión de la información. En este ejemplo, la información todavía está dispersa. A pesar de que hemos agrupado los datos, todavía no lo hemos hecho lo suficiente. Estamos mezclando información referida a conceptos diferentes dentro de la tupla *periodo*: la fecha de inicio y la fecha de finalización.

Cuando diseñamos un tipo de datos mediante tuplas estamos realizando una abstracción. Construimos un objeto complejo a partir de objetos más simples que englobamos con un único nombre. Esta abstracción se tiene que realizar de forma lógica:

Fijaos en que...

… una tupla es un objeto que contiene otros objetos, y éstos, a su vez, también pueden contener otros objetos.

Un periodo está formado por dos datos, fecha de inicio y fecha de finalización. Una fecha está formada por un día, un mes y un año.

```
tipo
    fecha = tupla
              dia: entero;
              mes: { enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre, noviembre, diciembre };
              año: entero;
            ftupla
    periodo = tupla
                inicio, fin: fecha;
              ftupla
ftipo
```

Reencontraremos el tema...

... de la abstracción de datos en el módulo "Introducción a la metodología descendente" para resolver problemas complejos.

La forma de operar con los datos será la misma que en el caso de las tuplas simples. Por ejemplo, la acción que nos muestra por la salida estándar una variable de tipo *periodo* podría ser:

```
accion escribir_periodo (ent p: periodo)
    { Pre: p.inicio.dia = D1 y p.inicio.mes = M1 y p.inicio.año = A1 y p.fin.dia = D2 y p.fin.mes = M2
      y p.fin.año = A2 }
    escribirEntero(p.inicio.dia);
    escribirCaracter("/");
    escribirCadena(p.inicio.mes);
    escribirCaracter("/");
    escribirEntero(p.inicio.año);
    escribirCaracter("–");
    escribirEntero(p.fin.dia);
    escribirCaracter("/");
    escribirCadena(p.fin.mes);
    escribirCaracter("/");
    escribirEntero(p.fin.año);
    { Post: hemos escrito en el canal estándar "D1/M1/A1 – D2/M2/A2" }
faccion
```

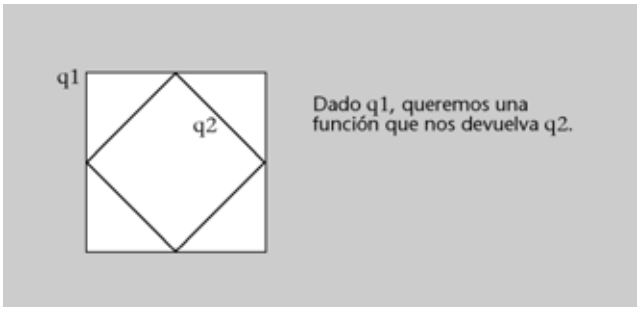
Lo que aparecerá en la salida estándar en este caso tendrá el siguiente formato: *día/mes/año – día/mes/año.*

Ejemplo 2

Podemos representar un cuadrado y un triángulo guardando las coordenadas de sus vértices:

```
tipo
    vertice = tupla
                x, y: real;
              ftupla
    cuadrado = tupla
                 a, b, c, d: vertice;
               ftupla
    triangulo = tupla
                 a, b, c: vertice;
               ftupla
ftipo
```

Veamos cómo sería una función que, dado un cuadrado, nos proporciona el cuadrado inscrito tal que sus vértices son los puntos medios de los lados del primero. Gráficamente:




```
funcion cuadradoInscrito (q: cuadrado): cuadrado
    var
    qNuevo: cuadrado;
    fvar
    { Pre: q.a.x = X1 y q.a.y = Y1 y q.b.x = X2 y q.b.y = Y2 y q.c.x = X3 y q.c.y = Y3 y q.d.x =
      = X4 y q.d.y = Y4 }
    qNuevo.a := calculoPuntoMedio(q.a, q.b);
    qNuevo.b := calculoPuntoMedio(q.b, q.c);
    qNuevo.c := calculoPuntoMedio(q.c, q.d);
    qNuevo.d := calculoPuntoMedio(q.d, q.a);
```

Observad la función *calculoPuntoMedio* en el punto 3.2.2 de este mismo módulo.

retorna qNuevo
{ Post: *qNuevo.a.x* = (*X1* + *X2*)/2 **y** *qNuevo.a.y* = (*Y1* + *Y2*)/2 **y**
qNuevo.b.x = (*X2* + *X3*)/2 **y** *qNuevo.b.y* = (*Y2* + *Y3*)/2 **y**
qNuevo.c.x = (*X3* + *X4*)/2 **y** *qNuevo.c.y* = (*Y3* + *Y4*)/2 **y**
qNuevo.d.x = (*X4* + *X1*)/2 **y** *qNuevo.d.y* = (*Y4* + *Y1*)/2 }
ffuncion

3.3.2. Tablas y tuplas

Consultad el esquema de recorrido aplicado a tablas al final del punto 2.4.1. de este mismo módulo.

Las tablas nos permiten guardar un número determinado de elementos del mismo tipo, pero en muchas ocasiones tenemos que tratar tablas que no tienen todas las posiciones completas. En estos casos, como hemos visto en el capítulo de tablas, necesitamos, de alguna forma, poder determinar cuál es el último elemento significativo de la tabla (que no tiene por qué ser la última posición de la tabla). Una de las técnicas que hemos comentado era la de utilizar una variable entera que nos indique hasta qué posición tenemos llena la tabla, o bien a partir de qué posición está vacía.

Un ejemplo muy claro de esto es el uso de una tabla para guardar los caracteres que forman una palabra. En este caso, no podemos fijar el número de caracteres que tendrán las palabras que gestionará nuestro algoritmo, ya que no todas son del mismo tamaño, aunque podemos suponer un tamaño máximo, por ejemplo, 40 caracteres. Esto nos permite dimensionar la tabla con el fin de que quepan todos los caracteres que forman la palabra, pero necesitamos, además, un entero que nos indique cuántas posiciones de la tabla contienen caracteres válidos.

De este modo, el tipo *palabra* estará formado por dos objetos, uno que guardará las letras de la palabra y otro que indicará cuál es la primera posición libre de la tabla (donde tenemos que guardar el próximo carácter de la palabra, si es que hay otros), o bien cuál es el último carácter significativo de la tabla:

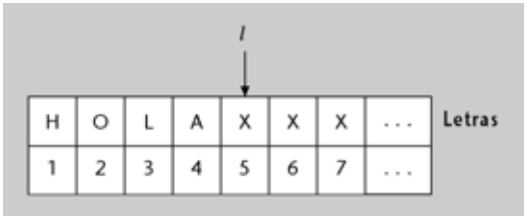
```
tipo
      palabra = tupla
                    letras: tabla [40] de caracter;
                    l: entero;
                ftupla
ftipo
```

Esta representación nos facilita el tratamiento de la tabla, ya que conocemos exactamente hasta qué posición de la tabla hay que tratar. También nos permite conocer la posición donde tenemos que guardar el siguiente carácter de la tabla en el momento en que la llenamos. Además, el hecho de utilizar una tupla permite que tengamos agrupados en un único objeto la tabla y la dimensión de ésta.

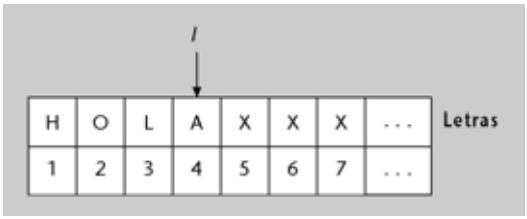
Por ejemplo, si guardamos la palabra de cuatro letras HOLA, tendríamos que la tabla contiene { “H”, “O”, “L”, “A”, “X”, “X”, “X”, … } donde “X” puede ser cualquier carácter.

El valor del campo *l* dependería de cuál de las dos opciones comentadas queremos utilizar.

En la primera opción, el campo *l* contendría el valor 5, que es la posición siguiente a la del último carácter válido de la tabla:



En la otra opción que hemos comentado, el campo *l* nos indicaría la última posición llena, en lugar de la primera vacía, o lo que es lo mismo, el número de caracteres significativos de la tabla. Por tanto, el valor del campo *l* sería 4, lo que indicaría que la tabla contiene 4 caracteres válidos. Gráficamente, esto sería:



De hecho, la diferencia entre los dos casos reside únicamente en qué valor inicial le damos al campo *l*, y en qué orden incrementamos y asignamos:

Caso 1

```
…
var
    p: palabra;
    c: caracter;
    esBlanco: booleano;
fvar
p.l := 1;
c := leerCaracter();
esBlanco := (c = ‘ ’);
mientras no esBlanco hacer
    p.letras[p.l] := c;
    p.l := p.l + 1;
    c := leerCaracter();
    esBlanco := (c = ‘ ’)
fmientras
…
```

Caso 2

```
…
var
    p: palabra;
    c: caracter;
    esBlanco: booleano;
fvar
p.l := 0;
c := leerCaracter();
esBlanco := (c = ‘ ’);
mientras no esBlanco hacer
    p.l := p.l + 1;
    p.letras[p.l] := c;
    c := leerCaracter();
    esBlanco := (c = ‘ ’)
fmientras
…
```

En el caso 1,...

... el campo *l* indica la primera posición vacía de la tabla. En el caso 2, indica la última posición llena, es decir, cuántos elementos significativos contiene la tabla.

Fijaos en el **caso 1**: el valor inicial del campo *l* es 1, que es la posición donde tenemos que guardar el próximo carácter, obtenemos un carácter del canal de entrada, lo guardamos en la posición indicada para el campo *l* y, a continuación, incrementamos este campo para que apunte a la siguiente posición. En el **caso 2**, en cambio, el valor inicial del campo *l* es 0 (la palabra no contiene ningún carácter), y justo antes de guardar un carácter en la tabla incrementamos *l*.

Normalmente, esta segunda representación será la que utilizaremos para hacer tratamientos, ya que nos resultará más práctico saber cuántos elementos significativos contiene una tabla que saber cuál es la primera posición vacía.

El tipo *palabra*...

... consta de una tabla de caracteres y de un entero que indica cuál es la posición del último carácter de la palabra dentro de la tabla.

Así pues, de ahora en adelante utilizaremos esta segunda representación para almacenar palabras. Por lo tanto, cuando utilicemos el tipo *palabra* a partir de ahora lo estaremos haciendo según esta representación y el tratamiento explicado en la segunda opción.

Ejemplo

```
tipo
    nota = { E, D, C, B, A };
    palabra = tupla
                letras: tabla [40] de caracter;
                l: entero;
              ftupla
    alumno = tupla
               codigoMatricula: entero;
               nombre: palabra;
               direccionE: palabra;
               notasPEC: tabla [5] de nota;
               nota: nota;
             ftupla
    grupo = tupla
              alumnos: tabla [75] de alumno;
              numAlumnos: entero;
            ftupla
ftipo
```

Fijaos en que...

... podemos tener tuplas de tablas, donde cada posición es, a su vez, una tupla.

En este caso, un grupo consta de una tabla, que contiene a los alumnos, y de un entero, que indica cuántos alumnos constan en la tabla.

Resumen

En este módulo hemos aprendido a utilizar los constructores de tipos para diseñar nuevos tipos de datos adecuados al problema concreto que es objeto de tratamiento, con el fin de aumentar la legibilidad y concreción de los algoritmos.

Para conseguirlo:

- En primer lugar, hemos visto cómo los tipos de datos que habíamos aprendido en los módulos anteriores son insuficientes para tratar determinados problemas que requieren objetos más estructurados, que se adapten mejor a la información que hay que representar en cada algoritmo concreto.

- Hemos aprendido cómo construir, a partir de los tipos de datos básicos y el constructor de tipos que habíamos visto en el módulo "Introducción a la algorítmica", dos nuevos tipos de datos estructurados: **tablas** y **tuplas**. Por tanto, lo que hemos hecho ha sido ampliar las posibilidades que ya conocemos del constructor de tipos.

  – Las tablas nos permiten agrupar elementos del mismo tipo, con la finalidad de realizar ciertos tratamientos que permiten aumentar la eficiencia de nuestros algoritmos

  – Las tuplas nos permiten agrupar elementos de tipos diferentes, con la finalidad de que el algoritmo resulte más legible.

- Hemos estudiado las características y posibilidades que ofrecían las tablas como agrupación de elementos para poder aplicar los esquemas de tratamiento secuencial aprendidos en el módulo "Tratamiento secuencial", el recorrido y la búsqueda. Y hemos visto también cómo las tablas nos abrían una nueva posibilidad mediante el acceso directo a sus elementos: hemos estudiado unos procedimientos de búsqueda y ordenación únicamente posibles con las tablas, ya que requieren acceso directo a los elementos.

- Hemos aprendido a definir tuplas para almacenar información relacionada entre sí. Esto nos ha permitido crear objetos que se componen de diferentes datos, que pueden ser de tipos diferentes y que, en conjunto, forman un único objeto estructurado.

- Hemos visto también cómo el constructor de tipos nos permite crear objetos tan estructurados como sea necesario, definidos a partir de tipos de datos que, a su vez, también pueden ser estructurados. Así pues, hemos visto cómo hay que definir tuplas que contienen tablas, tablas que contienen tuplas, tu-

plas que contienen tablas que a la vez contienen tuplas, etc., siempre con el objetivo de definir unos objetos en nuestro algoritmo que se adapten al máximo a la información que deben representar. Esto es lo que se denomina *hacer abstracción de datos*, puesto que estamos creando unos datos abstractos adaptados a la información del problema que el algoritmo tiene que resolver.

Tal como vimos en el módulo "Introducción a la programación", la construcción de un algoritmo deberá pasar por cuatro fases: entender el enunciado, plantear la solución, formularla y, finalmente, evaluarla. Vamos a ver qué nuevas posibilidades se nos ofrecen en algunas de estas fases con lo que hemos aprendido en este módulo.

1) Entender el enunciado.

Esta primera etapa consiste en definir claramente, mediante la especificación, **qué** es lo que tiene que hacer nuestro algoritmo. La especificación define **qué** hace el algoritmo, indicando con precisión cuál es el estado inicial de partida (precondición), y cuál el estado final después de ejecutarse el algoritmo (postcondición), siguiendo este esquema (que ya estudiamos en el módulo "Introducción a la programación"):

Definición de variables
{ Precondición: … }
Nombre del algoritmo
{ Postcondición: ... }

Así que, cuando llevamos a cabo la especificación, también definimos unas variables que utilizamos para definir los estados inicial y final. En este módulo hemos aprendido a definir y utilizar dos nuevos tipos de datos, tablas y tuplas y, por tanto, a partir de ahora deberemos tener en cuenta que en las especificaciones podremos definir y utilizar variables de estos tipos. Cuando nos planteemos **qué** debe hacer el algoritmo, lo podemos hacer en términos de tablas y tuplas si conviene.

2) Plantear el problema.

En este punto decidiremos si tenemos que utilizar datos estructurados para representar la información que tratará el algoritmo. El planteamiento que hay que hacer para decidir la utilización de tuplas y de tablas es muy diferente. En el primer caso, se trata básicamente de una cuestión de claridad o abstracción en la representación de la información, mientras que en el segundo, se debe claramente a cuestiones de tratamiento:

a) Tuplas. Debemos determinar si el algoritmo tiene que tratar objetos compuestos (objetos que no se pueden definir con los tipos de datos básicos). Si necesitamos representar algún objeto abstracto (no existente en los tipos básí-

cos del lenguaje algorítmico) formado por otros objetos, entonces tendremos que definir tuplas. De esta forma, podremos crear objetos a la medida de las necesidades del algoritmo.

b) Tablas. La utilización de tablas puede venir dada por diferentes necesidades:

- Necesitamos guardar, para su posterior tratamiento, los elementos de una secuencia de entrada. Tendremos que definir una tabla con la suficiente capacidad como para guardar los elementos que iremos obteniendo de la secuencia. El traspaso de los elementos de la secuencia a la tabla se llevará a cabo mediante un esquema de recorrido, si queremos traspasar todos los elementos de la secuencia, o de búsqueda, si sólo se tiene que traspasar una parte.
- Debemos ordenar un conjunto de elementos. Será necesario que guardemos estos elementos en la tabla para hacer después la ordenación utilizando el método de selección directa.
- Los datos que debemos tratar se adecuan a una representación en forma de tabla, ya que están formadas por un grupo de elementos del mismo tipo. Es el caso de vectores, matrices, palabras, frases, etc.

Si optamos por la utilización de alguna tabla en nuestro algoritmo, tendremos que decidir también si es necesario aplicarle algún esquema de tratamiento. Ya hemos visto que en una tabla se pueden aplicar los esquemas de tratamiento secuencial. En caso de que necesitemos hacer un tratamiento secuencial de los elementos de la tabla, tendremos que determinar cuál es el esquema adecuado (el de recorrido o el de búsqueda).

3) Formular la solución.

Una vez formulado el planteamiento del problema, podemos pasar a enunciar la solución mediante el lenguaje algorítmico. En este módulo hemos introducido nuevos elementos sintácticos para construir nuevos tipos de datos, por lo que se plantea la necesidad de tener en cuenta estos nuevos elementos en el apartado de definición de datos.

Si queremos crear objetos compuestos, será necesario definir las tuplas correspondientes, teniendo muy claro cuáles son los campos que debe tener la tupla para que se adecue a la información que debe representar. Ésta es la clave de la abstracción de datos: ser capaces de abstraer la información relevante del problema que queremos tratar.

La definición de las tablas es más sencilla, ya que sólo es necesario conocer el tamaño de la tabla. En algunos casos, sin embargo, definiremos tablas para guardar un número indeterminado de elementos, por lo que tendremos que conocer el número máximo de elementos posibles para dimensionar la tabla. Y tendremos que disponer de un control para saber cuántos elementos de la tabla son significativos. Hemos visto que podemos hacer esto definiendo una

tupla, que tiene como campos la tabla, y un entero, que indica cuántas posiciones de la tabla debemos tratar.

Una vez hayamos definido las variables del algoritmo, ya podremos pasar a refinar los esquemas de tratamiento que hayamos decidido en el planteamiento.

4) Evaluar la solución.

Como siempre, llegado este punto, es necesario repasar el algoritmo para comprobar que realmente hace lo que habíamos especificado en el primer punto. Estamos siguiendo una metodología de diseño que evita que podamos cometer muchos errores, pero debemos repasarlo para estar seguros de que hemos aplicado de forma totalmente correcta esta metodología, y que realmente el algoritmo se encuentra totalmente libre de errores.

En este módulo hemos introducido nuevos tipos de datos y, por tanto, deberemos comprobar que el algoritmo los utiliza de forma correcta.

En el caso de las tuplas, la comprobación es bastante trivial, básicamente tenemos que repasar que realmente sean adecuadas para los datos que representan. Es preciso comprobar que los campos de las tuplas son los que tienen que ser, no debemos agrupar datos que no tienen sentido juntos, ni dejar de agrupar los que claramente forman parte de un mismo objeto. También comprobaremos si el algoritmo utiliza campos de una tupla que no están en su definición, o si deja de utilizar alguno de los que se han definido.

Tanto en planteamiento del problema como en la formulación de la solución, tenemos que habernos planteado claramente qué abstracción de la información del problema hay que llevar a cabo para determinar los campos que debe tener cada tupla, por lo cual es difícil cometer errores en este punto, aunque a pesar de todo será importante repasarlo.

Para las tablas, tenemos que comprobar que están bien dimensionadas. Es decir, que el número de elementos que se ha indicado en la definición de la tabla será suficiente para introducir todos los que se guardarán después en el algoritmo.

Las tablas también introducen nuevos factores que hay que tener en cuenta al repasar los tratamientos. En los accesos a los elementos de las tablas hay siempre una variable como índice para determinar el elemento “actual” de la tabla. Debemos prestar especial atención al hecho de que esta variable no tome valores que no están comprendidos dentro de la dimensión de la tabla.

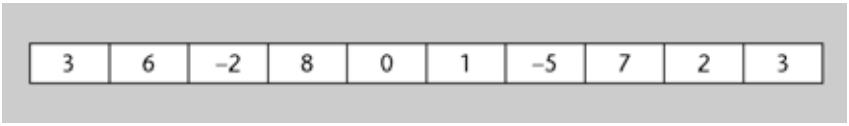
En los tratamientos secuenciales de las tablas, es importante prestar atención a los casos especiales que se derivan de la aplicación de los esquemas, y que ya vimos en el apartado homónimo del módulo anterior. La tabla puede no estar completamente llena, es necesario asegurarse de que se tratan los elementos

que hay que tratar y no más, haciendo un especial énfasis en el tratamiento del último elemento.

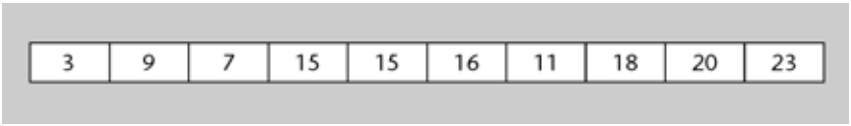
También puede darse el caso de que la tabla esté totalmente vacía, por lo que el tratamiento debe manejar correctamente este caso. El control del último elemento que trataremos se puede hacer mediante un centinela, o con un entero que indica la posición, pero en cualquier caso, hay que comprobar que en la aplicación del esquema no dejemos de tratar un elemento, o de lo contrario trabajemos con un elemento más de lo necesario. Por lo tanto, tendremos que comprobar que realmente la última posición tratada de la tabla es la que tiene que ser.

Ejercicios de autoevaluación

1. Dada una tabla de enteros de tamaño *N*, tendremos que obtener otra, también de tamaño *N*, que contenga las sumas parciales de los elementos de la primera. Es decir, el elemento *i*-ésimo de la segunda tabla tendrá que contener la suma desde el primer elemento hasta el *i*-ésimo de la primera tabla. Así, si tenemos la tabla:



deberemos obtener como resultado esta otra*:*



2. Se quiere construir una función que verifique que un código ISBN es correcto. El formato de un código ISBN (*International Standard Book Code*) es el siguiente:

Código de grupo (1 dígito)
Código del editor (4 dígitos)
Código del libro (4 dígitos)
Carácter/dígito de control (1 carácter/dígito)

Así, por ejemplo, el código ISBN de un libro de ingeniería es 0-6752-0993-5. La verificación del código se realiza a partir de aplicar una serie de operaciones sobre los nueve primeros dígitos, de cuyo resultado extraeréis el dígito/carácter de control. Los pasos para la obtención del control son los siguientes:

- Se multiplica cada uno de los dígitos por la posición que ocupa dentro del ISBN y se suman todas las multiplicaciones.
- Se divide el resultado por 11 y se guarda el resto. El resto tiene que ser entero.
- El resto anterior es el dígito de control. Si este resto es 10, se utiliza la letra X como control.

La función que nos piden recibirá como parámetros de entrada una tabla de nueve posiciones con el ISBN y un carácter que será el control. La función retornará un booleano indicando cierto si es un ISBN válido y falso en el caso contrario. Al realizar el algoritmo, podemos suponer que disponemos de una función *enteroACaracter*, que convierte un dígito en el carácter que representa este dígito. Así, *enteroACaracter*(3) nos retornaría el carácter '3'.

3. Dada una frase acabada en punto, debemos codificarla en una nueva frase siguiendo el procedimiento que exponemos a continuación: debemos tomar grupos de cinco caracteres e invertir el orden. Observad que el último grupo puede no tener cinco caracteres.

4. Disponemos de una secuencia de caracteres en minúscula por el canal de entrada que tenemos que codificar de acuerdo con un cierto código. Este código está almacenado en una tabla en la que, en cada posición, está guardado el código correspondiente a un carácter. Por tanto, en la primera posición de la tabla tenemos el código del carácter 'a', en la segunda el de la 'b', y así hasta la 'z'. Considerando que esta tabla *código* ya está inicializada con los códigos de todas las letras minúsculas del alfabeto, se trata de escribir un algoritmo en el que se codifique la secuencia que nos aparece por el canal de entrada y escribirla, codificada, en el canal de salida.

5. Aplicad la transposición de matrices a la encriptación de un texto. Supongamos que por el canal de entrada nos llega una frase, acabada en punto, con una longitud máxima de 100 caracteres. La transposición de texto se basa en ir poniendo los caracteres de esta frase en una tabla matriz 10×10, del modo que exponemos a continuación. Así, tenemos la frase:

"Esto es una frase acabada en punto."

Quedaría dispuesta en la matriz como:

E	S	T	O	β	E	S	β	U	N
A	β	F	R	A	S	E	β	A	C
A	B	A	D	A	β	E	N	β	P
U	N	T	O	β	β	β	β	β	β
β	β	β	β	β	β	β	β	β	β
.....									

Donde β es un carácter especial que introducimos para simbolizar el espacio en blanco entre dos palabras.

La transposición de texto se basa en generar una frase de salida a partir de haber recorrido la matriz anterior por columnas. Lo que haremos, al generar la frase de salida, es considerar tan sólo las filas en las que haya caracteres significativos. En el ejemplo, sólo las cuatro primeras.

6. Queremos diseñar todo lo que sea necesario para manipular números complejos expresados en notación binómica. Seguid estos pasos:

- definid un tipo de datos para estos números: un número complejo está formado por dos números reales, uno que representa la parte real y otro que representa la parte imaginaria.
- diseñad las operaciones para manipularlos:

– Crear números complejos: función en la que, dados dos números reales que representan la parte compleja y la parte imaginaria respectivamente, nos retorna un número complejo.

– Sumar números complejos: función en la que, dados dos números complejos, nos retorna el número complejo resultante de efectuar la suma.

– Multiplicar números complejos: función en la que, dados dos números complejos, nos retorna el número complejo resultante de efectuar el producto.

– Obtener la parte real de un número complejo: función en la que, dado un número complejo, nos retorna la parte real.

– Obtener la parte imaginaria de un número complejo: función en la que, dado un número complejo, nos retorna la parte imaginaria.

Recordemos el álgebra de los números complejos:

dados dos complejos *c1* y *c2*, siendo

$$c1 = r1 + i1 * i = (r1, i1) \text{ en forma binómica.}$$
$$c2 = r2 + i2 * i = (r2, i2) \text{ en forma binómica.}$$

la suma y el producto se definen como vemos a continuación:

$$c1 + c2 = (r1 + r2) + (i1 + i2) * i$$
$$c1 * c2 = (r1 * r2 - i1 * i2) + (r1 * i2 + i1 * r2) * i$$

7. Dado el tipo de datos *palabra* tal y como está definido en el apartado 3.3.2. de este módulo, diseñad las tres funciones/acciones siguientes:

a) Leer una palabra de una secuencia de entrada: dada una secuencia de caracteres (donde el próximo carácter puede ser un espacio en blanco o no) se deben retornar en un tipo *palabra* todos los caracteres diferentes de espacio de la secuencia hasta el próximo espacio, o hasta el final de ésta (el final vendrá dado por un carácter igual a '.').

b) Escribir una palabra en el canal de salida estándar: dado un parámetro de tipo *palabra*, la función debe escribir todos los caracteres que la forman en el canal de salida estándar.

c) Comparar dos palabras: dados dos parámetros de tipo *palabra*, retornar un valor booleano que indique si las dos palabras son iguales o no.

8. Se pide:
a) Definid un tipo de datos llamado *ligaFutbol* que guarde la información de los equipos y jugadores que participan en una liga de fútbol. La liga la pueden disputar hasta veinte equipos. La información que guardaremos por cada equipo será el nombre del equipo y la ciudad a la que pertenece (estos datos serán de tipo *palabra*); el número de partidos ganados, empatados y perdidos; el número de goles a favor y en contra, y los jugadores que hay en el equipo. Un equipo puede inscribir un máximo de 25 jugadores. De cada jugador se guardarán el nombre, el número de goles marcados, el número de tarjetas amarillas y rojas que se le han mostrado y el número de partidos jugados.

b) Diseñad una acción, en la que, dada como parámetro una variable de tipo *ligaFutbol*, que ya ha sido inicializada, escriba en el canal estándar los nombres y los puntos que tiene cada equipo de fútbol. El cálculo de puntos se hará sumando 1 punto por cada partido empatado y 3 puntos por cada partido ganado.

9. Una empresa que se dedica a fabricar 15 productos diferentes dispone de un almacén donde guarda todas las materias primas necesarias para la producción. Este almacén guarda un máximo de 200 materias primas diferentes, las cuales están identificadas con un código numérico (asignados de forma consecutiva comenzando por 1). Además del código, cada materia tiene una referencia (de tipo *palabra*) y un precio de coste. También será básico saber qué cantidad se encuentra en *stock* de cada materia prima en el almacén.

Cada producto fabricado consta de un conjunto de materias primas (como máximo, de 10), y de cada una de éstas habrá una determinada cantidad. Además, los productos tendrán un coste de fabricación y un nombre (de tipo *palabra*). El coste final de cada producto será la suma del coste de fabricación más el coste de cada una de las materias primas que lo componen para las cantidades de cada una. Se os pide:

a) Diseñad los tipos de datos necesarios para guardar toda la información.

b) Diseñad una función que, dado un producto y un almacén, nos calcule el coste del producto con los datos de éste y del almacén dado.

Solucionario

1. Especificamos primero el enunciado del problema:

t, sp: **tabla** [N] **de entero**
{ Pre: $N > 0$ }
sumasParciales
{ Post: Cada uno de los elementos de la tabla *sp* contiene la suma de los elementos de la tabla *t* desde el primero hasta el que ocupa la misma posición del elemento en *sp*. }

Veamos cómo podemos plantear la solución:

Se trata de efectuar un recorrido sobre la tabla sp, de manera que, en cada iteración del recorrido se obtenga la suma parcial correspondiente. La variable índice de este recorrido nos indicará qué suma parcial estamos considerando en la actualidad. Ahora bien, para obtener esta suma parcial podemos pensar en hacerlo de las dos formas siguientes:

- El valor de esta variable índice nos indica que tenemos que sumar, de la tabla *t*, todos los elementos desde el primero hasta el que ocupa la posición determinada, precisamente, por este índice. Esto es, claramente, otro recorrido.

- La otra posibilidad se basa en aprovechar que la suma parcial de los $i + 1$ primeros elementos es igual a la suma parcial de los i primeros elementos más el elemento de la tabla que ocupa la posición $i + 1$.

Aprovecharemos esta segunda opción evitando, así, hacer el segundo recorrido. Por lo que respecta al recorrido con el que se llenará la tabla *sp*, sus partes se concretarían de la siguiente forma:

- *prepararSecuencia*: inicialización de la variable índice para indicar que tenemos que tratar el primer elemento.

- *tratamientoInicial*: a partir de la manera de calcular las sumas parciales tenemos que notar que el primer elemento merece una atención especial. El cálculo de cada suma parcial se basa en una suma parcial anterior ya calculada. En el caso de la primera, no tendremos esta suma parcial anterior, y la suma parcial que nos interesa coincide con el valor del primer elemento de la tabla *t*. Así pues, este tratamiento distintivo del primer elemento es el que haremos en el *tratamientoInicial*.

- *tratarElemento*: se trata de calcular la suma parcial correspondiente a la posición que establece el índice a partir de la suma parcial de la posición anterior más el elemento de la tabla *t* que ocupa la posición actual.

- *avanzarSecuencia*: corresponde a incrementar la variable índice.

- *finSecuencia*: corresponde a verificar que la variable índice todavía no ha llegado al límite superior establecido en la declaración (*N* en este caso).

Podemos pasar ahora ya a escribir el algoritmo:

algoritmo sumasParciales

```
const
    N= 10;
fconst
var
    t, sp: tabla [N] de entero;
    i : entero;
fvar

{ Pre: N > 0 }
i := 2;
sp[1] := t[1];
mientras no i > N hacer
    sp[i] := sp[i – 1] + t[i];
    i := i + 1
fmientras
Post:{ cada uno de los elementos de la tabla sp contiene la suma de los elementos de la
tabla t desde el primero hasta el que ocupa la misma posición del elemento en sp }
```

falgoritmo

Si utilizásemos **para ... fpara**, quedaría así:

```
sp[1] := t[1];
para i := 2 hasta N hacer
    sp[i] := sp[i–1] + t[i]
fpara
```

2. Especificamos primero esta función:

funcion esISBNValido (codigoISBN: **tabla**[9] **de entero**, control: **caracter**): **booleano**

{ Pre: *codigoISBN* es una tabla de enteros y *control* es un carácter con una *X* o el carácter correspondiente a los enteros 1, ..., 9 }

{ Post: *esISBNValido* será **cierto** si el control de los primeros 9 dígitos del ISBN contenidos en *codigoISBN* coincide con *control*. En caso contrario, *ISBNValido* será **falso** }

El planteamiento de esta función obedece claramente a un recorrido de la tabla ISBN en la que realizaremos el primer paso indicado en el cálculo de control, y un tratamiento final en el que se acaba de calcular este control y se verifica si se corresponde o no con el control indicado. Así pues, si pensamos en un esquema de recorrido, la adaptación a la resolución de este problema sería como veremos a continuación:

- *prepararSecuencia*: inicializar el valor del índice.
- *tratamientoInicial:* inicializar una variable en la que tendremos que ir acumulando la suma que hay que realizar durante el recorrido.
- *tratarElemento*: multiplicar el elemento de la posición indicada mediante el valor actual del índice por el valor mismo del índice. Esta multiplicación se tiene que sumar a las multiplicaciones ya hechas.
- *avanzarSecuencia*: avanzar el índice una posición. En este caso, será incrementar la variable entera que actúe de índice.
- *finalSecuencia*: verificar que el índice no sea mayor que 9.
- *tratamientoFinal*: acabar de hacer el cálculo del control y compararlo con el control que se ha pasado a la función como parámetro.

En lenguaje algorítmico, esta función podría quedar de la siguiente forma:

```
funcion esISBNValido (codigoISBN: tabla[9] de entero, control: caracter): booleano

    { Pre: codigoISBN es una tabla de enteros y control es un carácter con una "X" o el carácter
      correspondiente a los enteros 1, ..., 9 }

    var
        indice, suma, resto: entero;
        carControl: caracter;
        valido: booleano;
    fvar
    indice := 1;
    suma := 0;
    mientras indice ≤ 9 hacer
        suma := suma + indice * codigoISBN[indice];
        indice := indice + 1
    fmientras
    resto := suma mod 11;
    si resto < 10 entonces
        carControl := codigoACaracter (resto + caracterACodigo ('0'))
    sino
        carControl := 'X'
    fsi
    si carControl = Control entonces
        valido := cierto
    sino
```

```
        valido := falso
    fsi
    retorna valido;
    { Post: esISBNValido será cierto si el control de los primeros 9 dígitos del ISBN contenidos
      en codigoISBN coinciden con control. En caso contrario esISBNValido será falso }
ffuncion
```

Podemos sustituir el bucle **mientras ... fmientras** por un bucle **para ... fpara** como el que se presenta a continuación:

```
para indice := 1 hasta 9 hacer
    suma := suma + indice * codigoISBN[indice]
fpara
```

3. Especificamos el problema:

{ Pre: en el canal de entrada tenemos una frase acabada en '.' }
codifica5en5
{ Post: en el canal de salida generamos una frase que se corresponde con la leída por el canal de entrada, de manera que los grupos de 5 caracteres de la salida se corresponden con el mismo grupo que en la entrada pero con el orden invertido. Estas agrupaciones no son aleatorias, sino que se corresponden con agrupaciones de 5 caracteres correlativas desde el inicio de la frase }

En este problema se hace un recorrido de la frase que iremos leyendo del canal de entrada. Podemos dividir el tratamiento en tres partes:

- Obtener un grupo de cinco caracteres. Se debe prever la posibilidad de que el último grupo no sea completo (si la longitud de la frase no es un múltiplo de 5).
- Invertir el orden de este grupo.
- Escribir el grupo ya codificado en el canal de salida.

Estas operaciones, que hay que realizar con un grupo de cinco caracteres, quedan simplificadas si disponemos cada grupo en una tabla. De este modo, los tres puntos anteriores se pueden reescribir de manera más concisa, como:

- Leer un grupo de caracteres y ponerlo en una tabla.
- Invertir el orden de esta tabla.
- Escribir esta tabla en el canal de salida.

Ya hemos visto en un ejemplo la operación de invertir el orden de los elementos de una tabla. Por tanto, podemos pensar en realizar una función que nos retorne la tabla con el orden invertido. La función *invTabla* respondería a una especificación como la siguiente:

funcion *invTabla*(t: **tabla** [5] **de caracter**, long: **entero**): **tabla** [5] **de caracter**

{ Pre: $long \leq 5$, $t = T$ }
{ Post: para i desde 1 hasta $long$, $t[i] = T[long - i + 1]$ }

Y se correspondería con el algoritmo visto en el ejemplo del módulo. Así pues, con la ayuda de esta función, el recorrido que hay que realizar de la frase que leemos por el canal de entrada sería:

- *prepararSecuencia*: tendremos que leer un elemento de la secuencia (ya que estamos detectando el final mediante una marca).
- *tratarElemento*: se correspondería con la realización de las tres tareas anteriores.
- *avanzarSecuencia*: lo asociamos con la obtención de un nuevo grupo de caracteres, en el cual ya se va leyendo del canal de entrada.
- *finalSecuencia*: verificar si el elemento leído es un '.'

El algoritmo final podría quedar como vemos a continuación:

```
algoritmo codifica5en5

    var
        t: tabla [5] de caracter;
```

```
        long, i: entero;
        c: caracter;
    fvar

    { Pre: en el canal de entrada tenemos una frase acabada en '.' }
    c := leerCaracter();
    mientras no (c = '.') hacer
        long := 0;
        mientras no (c = '.') y (long < 5) hacer
                long := long + 1;
                t[long] := c;
                c := leerCaracter()
        fmientras
        t := invTabla(t, long);
        para i := 1 hasta long hacer
            escribirCaracter(t[i])
        fpara
fmientras

    { Post: en el canal de salida generamos una frase que se corresponde con la leída por el
      canal de entrada de manera que los grupos de 5 caracteres de la salida se corresponden
      con el mismo grupo que en la entrada, pero con el orden invertido. Estas agrupaciones
      no son aleatorias, sino que se corresponden con agrupaciones de 5 caracteres correlati-
      vos desde el inicio de la frase }

falgoritmo
```

4. Como siempre, especificamos primero el problema:

```
codigo: tabla[26] de caracter
{ Pre: en el canal de entrada disponemos de una frase acabada en '.' y la tabla codigo está llena
  con la codificación correspondiente a cada uno de los 26 caracteres de la 'a' a la 'z'. La co-
  dificación de la 'a' la tendremos en la primera posición, la de la 'b' en la segunda, y así su-
  cesivamente }
codifica
{ Post: en el canal de salida hemos escrito una frase de la misma longitud que la leída por el
  canal de entrada, donde cada carácter es el código, de acuerdo con la tabla codigo, del co-
  rrespondiente carácter en la frase de entrada }
```

De la misma manera que en el ejercicio anterior, en este caso tenemos que realizar un recorrido de la frase que está situada en el canal de entrada. En este caso, sin embargo, deberemos utilizar un tratamiento diferente: codificar el carácter leído. Dado que la tabla de codificación *código* esta indicada por la posición que ocupa el carácter en el alfabeto, esta codificación se podrá efectuar accediendo directamente al código del carácter actual, de la misma forma que hemos actuado en el ejemplo en que contábamos las frecuencias de aparición de un carácter en un texto.

El recorrido nos quedará, entonces, como sigue:

- *prepararSecuencia*: tendremos que leer un elemento de la secuencia (ya que estamos detectando el final mediante una marca).
- *tratarElemento*: obtener la codificación del carácter actual, y escribir esta codificación por el canal de salida.
- *avanzarSecuencia*: leer el siguiente elemento del canal de entrada.
- *finalSecuencia*: verificar si el elemento leído es un '.'

El algoritmo final es:

```
algoritmo codifica

var
    codigo: tabla [26] de caracter
    codigoA, pos: entero;
    codificacion, c: caracter;
fvar

{ Pre: en el canal de entrada tenemos una frase acabada en '.' y la tabla codigo contiene la co-
  dificación correspondiente a cada uno de los 26 caracteres de la 'a' a la 'z' }
codigoA := caracterACodigo('a');
c := leerCaracter();
mientras no c = '.' hacer
```

```
        pos := caracterACodigo(c) – codigoA + 1;
        codificacion := codigo[pos];
        escribirCaracter(codificacion);
        c := leerCaracter()
    fmientras

    { Post: en el canal de salida hemos escrito una frase de la misma longitud que la leída por el
      canal de entrada y donde cada carácter es el código, de acuerdo con la tabla codigo, del co-
      rrespondiente carácter en la frase de entrada }
```

5. Como siempre, especificamos primero el problema:

```
{ Pre: en el canal de entrada tenemos una frase acabada en punto y de una longitud máxima
  de 100 caracteres }
transposicionDeTexto
{ Post: en el canal de salida hemos escrito una frase que resulta de aplicar la transposición de
  texto según una matriz de 10 filas y 10 columnas en la frase leída por el canal de entrada }
```

Hagamos un primer acercamiento al problema, que podemos dividir en dos partes:

1. Llenar la matriz a partir de la frase leída del canal de entrada.

2. Generar la salida a partir de recorrer por columnas la matriz en la que tenemos almacenada la frase.

La primera parte se corresponde, de manera clara, con un recorrido del canal de entrada en el que el tratamiento que hay que hacer a cada uno de los caracteres es almacenarlo en la posición correspondiente de la matriz. Por este motivo, nos será necesario mantener un índice para las filas y uno para las columnas, que se tendrán que ir incrementando a medida que vayamos recorriendo el canal de entrada. Además, después de este recorrido deberemos tener anotada cuál es la última fila y la última columna en las que tenemos un carácter significativo (hasta qué fila tenemos llena la matriz).

En segundo lugar, la generación de la frase de salida se basará en un recorrido de la matriz. Este recorrido tendrá que tratar, en cada paso, una columna de la matriz, siendo el tratamiento, a su vez, un recorrido de la columna, de modo que escribiremos cada uno de los caracteres que se encuentran en sus posiciones en el canal de salida.

Veamos con un poco más de detalle estos recorridos. Por lo que respecta al de la primera parte, tendremos lo siguiente:

- *tratamientoInicial*: inicializar los valores de los índices de la tabla.
- *tratarElemento*: asignar el elemento leído a la posición referenciada por los dos valores del índice. También será necesario actualizar los índices de la tabla de manera que ésta se vaya llenando por filas. Notamos que esta actualización no la hacemos en el *avanzarSecuencia*, ya que el recorrido que planteamos es un recorrido de la secuencia que tenemos en el canal de entrada.
- *avanzarSecuencia*: leer el siguiente carácter del canal de entrada.
- *finalSecuencia*: verificar que el carácter leído no es la marca de final de secuencia.
- *tratamientoFinal*: determinar cuáles son los valores de la última fila y columna en las que hemos escrito un carácter.

En cambio, el segundo recorrido se corresponde con el recorrido de una tabla bidimensional. Este recorrido lo hacemos por columnas.

- *prepararSecuencia*: inicializar el valor del índice correspondiente a las columnas.
- *tratarElemento*: aquí, cada elemento será una columna de la tabla, la cual tendremos que recorrer hasta la última fila en que hemos escrito. Este tratamiento será, por tanto, un recorrido de un vector fila:
  - *prepararSecuencia*: inicializar el valor del índice de filas.
  - *tratarElemento*: escribir en el canal de salida el elemento referenciado de manera conjunta por el índice de columnas y el de filas.

  - *avanzarSecuencia*: incrementar el índice de filas.
  - *finalSecuencia*: verificar que la variable índice no sobrepase la máxima fila en la que tenemos valores significativos.

- *avanzarSecuencia*: incrementar el índice de columnas.
- *finalSecuencia*: verificar que la variable índice no sobrepase la máxima columna en la que tenemos valores significativos.

Ahora ya podemos escribir el algoritmo completo:

```
algoritmo TrasposicionDeTexto

    var
    t: tabla [10,10] de caracter;
    fila, col: entero;
    maxFila, maxCol: entero;
    c: caracter;
    fvar

    { Pre: en el canal de entrada tenemos una frase acabada en punto de una longitud máxima de 100 caracteres }

    fila := 1;
    col := 1;
    c := leerCaracter();
    mientras c ≠ '.' hacer
        t[fila, col] := c;
        col := col + 1;
        si col > 10 entonces
            col := 1;
            fila := fila + 1
        fsi
        c := leerCaracter()
    fmientras
    maxFila := fila;
    si maxFila > 1 entonces
        maxCol := 10
    sino
        maxCol := col
    fsi
    col := 1;
    mientras col ≤ maxCol hacer
        fila := 1;
        mientras fila ≤ max Fila hacer
            escribirCaracter(t[fila,col])
            fila := fila + 1
        fmientras
        col := col + 1
    fmientras
falgoritmo
{ Post: en el canal de salida hemos escrito una frase que resulta de aplicar la transposición de texto según una matriz de 10 filas y 10 columnas a la frase leída por el canal de entrada }
```

Podemos hacer el segundo recorrido con un bucle **para ... fpara**:

```
para col := 1 hasta maxCol hacer
    para fila := 1 hasta maxFila hacer
        escribirCaracter(t[fila, col])
    fpara
fpara
```

6.

a) Un número complejo está formado por dos números reales; por tanto, tenemos que definir un nuevo tipo de datos que nos tiene que permitir agrupar dos objetos de tipo real. Debemos definir, pues, el tipo de datos *complejo* como una tupla con dos campos de tipo real:

```
    tipo complejo = tupla
                        re, im : real
                    ftupla
    ftipo
```

b) En este apartado tenemos que diseñar un conjunto de funciones. Para cada una tendremos que seguir las pautas de diseño:

- Función *crear*

  Entendemos el enunciado, pasamos a especificar lo que nos pide el enunciado que debe llevar a cabo esta función:

  funcion crear(x, y: **real**): **complejo**
  { Pre: $x = X$ **e** $y = Y$ }
  { Post: *crear*(x, y).*re* = X **y** *crear*(x, y).*im* = Y }

  Planteamos el problema: la función sólo tiene que asignar los valores que recibe como parámetro a cada uno de los dos campos de una variable auxiliar de tipo *complejo* que será retornada como resultado de la función.

  Formulamos la solución: escribimos en lenguaje algorítmico lo que hemos indicado en el apartado anterior.

  ```
  funcion crear (x, y: real): complejo
      var
          c: complejo;
      fvar
      c.re := x;
      c.im := y;
      retorna c
  ffuncion
  ```

  Evaluamos la solución: es evidente que la función hace exactamente lo que se ha especificado en el apartado 1, las dos asignaciones hacen que el complejo retornado contenga en sus dos campos los valores dados como parámetro.

- Función *sumar*

  Entendemos el enunciado:

  funcion sumar(c1, c2: complejo): **complejo**
  { Pre: $c1 = C1$ **y** $c2 = C2$ }
  { Post: *sumar*($c1$, $c2$).*re* = $C1.re + C2.re$ **y** *sumar*($c1$,$c2$).*im* = $C1.im + C2.im$ }

  Planteamos el problema: la función debe sumar los valores de las partes reales y después los de las partes imaginarias de los dos parámetros, y asignar los resultados a los campos real e imaginario de una variable auxiliar de tipo complejo que será retornada como resultado de la función.

  Formulamos la solución:

  ```
  funcion sumar (c1, c2: complejo): complejo
      var
          c: complejo;
      fvar
      c.re := c1.re + c2.re;
      c.im := c1.im + c2.im;
      retorna c
  ffuncion
  ```

  Evaluamos la solución: la función hace exactamente lo que se ha especificado en el apartado 1, las dos asignaciones hacen que el complejo retornado contenga en sus dos campos las sumas de los campos correspondientes de los dos complejos dados por parámetro.

- Función *multiplicar* (para no extendernos inútilmente, obviaremos algunos apartados porque son muy evidentes).

  Entendemos el enunciado:

  funcion multiplicar (c1, c2: complejo): **complejo**
  { Pre: $c1 = C1$ **y** $c2 = C2$ }
  { Post: *multiplicar*($c1$, $c2$).*re* = ($C1.re * C2.re$) – ($C1.im * C2.im$) **y**
  multiplicar($c1$, $c2$).*im* = ($C1.re * C2.im$) + ($C1.im * C2.re$) }

Formulamos la solución:

```
funcion multiplicar(c1, c2: complejo): complejo
    var
        c: complejo;
    fvar
    c.re := (c1.re * c2.re) – (c1.im * c2.im);
    c.im := (c1.re * c2.im) + (c1.im * c2.re);
    retorna c
ffuncion
```

- Función *parteReal*

Entendemos el enunciado:

```
funcion parteReal (c: complejo): real
    { Pre: c = C }
    { Post: parteReal(c) = C.re }
```

Formulamos la solución:

```
funcion parteReal(c: complejo): real
    retorna c.re
ffuncion
```

- Función *parteImaginaria*

Entendemos el enunciado:

```
funcion parteImaginaria(c: complejo): real
    { Pre: c = C }
    { Post: parteImaginaria(c) = C.im }
```

Formulamos la solución:

```
funcion parteImaginaria(c: complejo): real
    retorna c.im
ffuncion
```

7. Seguiremos las pautas de diseño para cada uno de los tres apartados; en todos utilizaremos el tipo *palabra*, tal y como se ha definido en los apuntes:

```
tipo
    palabra = tupla
                  letras: tabla [MAX] de caracter;
                  l: entero;
              ftupla
ftipo
```

Utilizamos la constante MAX,...

... que determinará la medida máxima de las palabras que trataremos. Tendrá que estar declarada en la sección de constantes.

a) Leer palabra

Entendemos el enunciado: será necesario realizar una función, ya que hay que retornar un resultado.

```
funcion leerPalabra(): palabra
{ Pre: leeremos una secuencia de caracteres de la entrada estándar }
{ Post: leerPalabra.l = 0 si no leemos ningún carácter diferente de espacio, sino leerPalabra.letras contiene los caracteres diferentes de espacio que se han leído hasta el primer espacio posterior a algún carácter diferente de espacio, y leerPalabra.l es igual que el número de caracteres diferentes de espacio que se han leído }
```

Planteamos el problema: para empezar, habrá que saltar todos los espacios, por lo que comenzaremos con un esquema de búsqueda para localizar el primer carácter diferente de espacio. Después tendremos que hacer un segundo esquema de búsqueda hasta encontrar el próximo espacio o hasta que se lea un punto, guardando en la tabla los caracteres leídos, contándolos al mismo tiempo y controlando que no superemos la capacidad de la tabla.

Formulamos la solución: escribimos en lenguaje algorítmico lo que hemos indicado en el apartado anterior.

```
funcion leerPalabra(): palabra
    var
        p: palabra;
        c: caracter;
    fvar
    c := leerCaracter();
    mientras c = ' ' hacer
        c := leerCaracter()
    fmientras
    p.l := 0;
    mientras (c ≠ '.') i (c ≠ ' ') hacer
        si (p.l < MAX) entonces
            p.l := p.l + 1;
            p.letras[p.l] := c
        fsi
        c := leerCaracter();
    fmientras
        retorna p
ffuncion
```

Evaluamos la solución: la primera búsqueda lee tantos espacios como haya, hasta que se encuentre un carácter diferente de espacio. Acabada esta búsqueda, la variable *c* contiene un carácter diferente de espacio. La segunda búsqueda continúa leyendo caracteres hasta que llegue el próximo espacio o un punto. Si *c* ya era un punto (la palabra está vacía) retornamos una palabra que tiene el campo *l* a 0; por tanto, el caso en que no hay palabra está bien tratado. El campo *l* se inicializa a 0 y se incrementa siempre justo antes de guardar un carácter en la tabla, por lo que guardará siempre el número total de caracteres de la tabla. Cuando *l* es igual a *MAX*, ya no guardamos ningún carácter más en la tabla; por tanto, en caso de que la palabra sea mayor que la capacidad de la tabla, no guardaremos los caracteres que no caben, y el índice no saldrá de rango. ¡La solución es correcta!

b) Escribir palabra.

Entendemos el enunciado: se tratará de una acción, ya que no se tiene que retornar ningún resultado.

```
accion escribirPalabra (ent p: palabra)

{ Pre: p = P }
{ Post: hemos escrito en la salida estándar los P.l primeros caracteres de la tabla P.letras }
```

Planteamos el problema: tendremos que realizar un recorrido de la tabla *p.letras* escribiendo en la salida estándar cada elemento de esta tabla, desde el elemento 1 hasta el elemento *p.l*.

Formulamos la solución:

```
accion escribirPalabra (ent p: palabra)
    var
        i: entero;
    fvar
    para i := 1 hasta p.l hacer
        escribirCaracter(p.letras[i])
    fpara
faccion
```

Evaluamos la solución: la composición iterativa **para** nos asegura que el esquema trata los *p.l* primeros elementos de la tabla.

c) Palabras iguales

Entendemos el enunciado: habrá que realizar una función, al ser necesario retornar un resultado de tipo **booleano**.

```
funcion palabrasIguales (p1, p2: palabra): booleano
{ Pre: p1 = P1 y p2 = P2 }
{ Post: palabrasIguales(p1, p2) será cierto si P1.l = P2.l y los P1.l primeros elementos de P1.letras
son iguales a los P2.l primeros elementos de P2.letras, y falso en caso contrario }
```

Planteamos el problema: como indica la postcondición, si los campos *l* de las dos palabras son diferentes, no tendremos que comprobar nada más, el resultado es falso. Si son iguales, enton-

ces tendremos que seguir un esquema de búsqueda hasta encontrar el primer carácter diferente entre las dos tablas. Si no encontramos este carácter, las dos tablas son iguales.

Formulamos la solución: escribimos en lenguaje algorítmico lo que hemos indicado en el apartado anterior.

```
funcion palabrasIguales (p1, p2: palabra): booleano
    var
        iguales: booleano;
        j: entero;
    fvar
    iguales := ( p1.l = p2.l );
    si iguales entonces
        j := 1
        mientras (j < p1.l) y (p1.letras[j] = p2.letras[j]) hacer
            j := j + 1
        fmientras
        iguales := (p1.letras[j] = p2.letras[j])
    fsi
    retorna iguales
ffuncion
```

Evaluamos la solución: si las dos tablas contienen el mismo número de elementos, el esquema de búsqueda nos encuentra el primer carácter que sea diferente, si hay alguno. Cuando salimos de la composición iterativa, estamos en el carácter diferente, o en el último (*p1.l*) que debemos comparar. Por tanto, el tratamiento final efectúa la comprobación del último carácter, lo que nos asegura que tratamos todos los caracteres de las palabras.

8.

a)

```
tipo

    tJugador = tupla
                   nombre: palabra;
                   numAmarillas, numRojas: entero;
                   partidosJugados: entero;
               ftupla

    tEquipo = tupla
                   nombre: palabra;
                   ciudad: palabra;
                   partidosGanados, partidosPerdidos: entero;
                   partidosEmpatados: entero;
                   golesFavor, golesContra: entero;
                   jugadores: tupla
                                  jugador: tabla [25] de tJugador;
                                  num: entero;
                              ftupla
               ftupla
    ligaFutbol = .tupla
                   equipos: tabla [20] de tEquipo;
                   num: entero;
               ftupla
ftipo
```

b)

Entendemos el enunciado: especificamos la acción solicitada:

```
accion escribirPuntos (ent l: ligaFútbol)

{ Pre: l = L }
{ Post: hemos escrito en el canal estándar el campo nombre de los L.num elementos de la tabla
  L.equipos junto con los puntos de cada uno. Los puntos de un equipo se calcularán a partir
  de los campos partidosEmpatados y partidosGanados de la variable de tipo tEquipo, con la
  fórmula partidosEmpatados + partidosGanados * 3 }
```

Planteamos el problema: se trata de efectuar un recorrido, ya que tenemos que tratar todos los *l.num* equipos de la tabla, y para cada uno deberemos escribir el campo nombre y la suma de los campos *partidosEmpatados* y el campo *partidosGanados* multiplicado por 3.

Formulamos la solución: escribimos en lenguaje algorítmico lo que hemos dicho en el apartado anterior.

```
accion escribirPuntos (ent l: ligaFutbol)
    var
    i: entero;
    fvar
    para i := 1 hasta l.num hacer
        escribirPalabra(l.equipos[i].nombre);
        escribirCaracter(':');
        escribirEntero(l. equipos[i].partidosEmpatados + l.equipos[i].partidosGanados * 3);
        escribirCaracter(' ')
    fpara
faccion
```

Utilizamos la acción escribirPalabra vista en el ejercicio anterior.

Evaluamos la solución: la composición iterativa **para** nos asegura que el esquema trata todos los elementos de la tabla.

9.

a) Necesitamos, por un lado, guardar todas las materias primas en alguna estructura. Puesto que conocemos el número máximo de materias que hay en el almacén, el almacén puede ser una tupla con una tabla, donde cada elemento será una materia, y un entero que indique cuántas materias hay. La posición que ocupa una materia dentro de la tabla puede coincidir con el código de la materia (ya que nos indican que se asigna de forma consecutiva comenzando por 1). Así, en la posición 1 de la tabla estará la materia con código 1, etc. Esto nos ahorra guardar el código de la materia y nos facilita la búsqueda de materias dentro de la tabla, ya que el mismo código nos determina la posición donde se encuentra la materia que buscamos. Una materia prima será una tupla porque nos interesa guardar varios componentes: referencia, precio de coste y *stock*. Por otra parte, los productos fabricados también los tendremos que guardar en una tabla, y cada producto será una tupla que contendrá el nombre del producto, el coste de fabricación y la composición. La composición de un producto nos tiene que indicar qué materias primas contiene (sólo necesitamos el código, ya que el resto de la información se encuentra en el almacén), y qué cantidad de cada una. Realizamos una tupla con estas dos informaciones, de modo que la composición será una tabla que contendrá como tipo de elemento esta tupla. Así pues, tendremos:

```
tipo
    materia = tupla
                referencia: palabra;
                precioCoste, stock: entero;
              ftupla
    producto = tupla
                 nombre: palabra;
                 costeFabricacion: entero;
                 composicion: tupla
                                materias: tabla [10] de tupla
                                                       codigoMat: entero;
                                                       cantidad: entero;
                                                     ftupla
                                num: entero;
                              ftupla
               ftupla
    almacen = tupla
                 materias: tabla [200] de materia;
                 numMat: entero;
              ftupla
    productos = tabla [15] de producto;
ftipo
```

b) Entendemos el enunciado y especificamos la función pedida:

```
funcion costeProducto (p: producto; m: almacen): entero
```

{ Pre: $p = P$ **y** $m = M$ }
{ Post: la función *costeProducto* retorna el coste calculado como P.costeFabricación + la suma del coste de las *P.composicion.num* materias de *P.composicion.materias* multiplicadas por la cantidad de cada una }

Planteamos el problema: el coste del producto es la suma de dos factores, el coste de fabricación y el coste de las materias. El primero es trivial, ya que está contenido en la variable producto. El segundo se tiene que calcular haciendo un recorrido por todas las materias primas del pro-

ducto, y multiplicando el coste de cada una por la cantidad que contiene el producto. El coste de la materia prima lo tenemos en el campo *materia* de *almacén*. Por tanto, buscaremos en esta tabla cada materia prima en la posición indicada por los códigos que tenemos en el *producto*. Se trata de un recorrido por el que conocemos el número de elementos que trataremos, ya que la variable *producto* contiene un campo que nos indica de cuántas materias está formado.

Formulamos la solución: escribimos en lenguaje algorítmico como hemos indicado en el apartado anterior.

```
funcion costeProducto (p: producto; m: almacen): entero
    var
        c, j: entero;
    fvar
    c := p.costeFabricacion;
    para j := 1 hasta p.composicion.num hacer
        c := c + (m.materias[p.composicion.materias[j].codigoMat].precioCoste
            * p.composicion.materias[j].cantidad)
    fpara
retorna c
ffuncion
```

Evaluamos la solución: el valor inicial de la variable c es igual al coste de fabricación del producto; suponiendo que el producto no contenga ninguna materia prima, éste sería el valor retornado. La composición iterativa **para** nos asegura que trataremos todas las materias primas que contiene el producto, y para cada una sumaremos su coste multiplicado por la cantidad que contiene el producto.

Glosario

acceso directo
Posibilidad de acceder a un elemento de la tabla sin la necesidad de pasar antes por todos los predecesores.

centinela
Valor no significativo, que se guarda en la posición siguiente al último valor significativo de una tabla, para indicar hasta dónde llegan los valores significativos.

índice
Conjunto de valores que sirve para referenciar cada uno de los elementos de una tabla.

matriz
Tabla bidimensional.

tabla
Tipo de datos estructurado, homogéneo, de acceso directo y dimensión fija.

tupla
Tipo de datos estructurado, heterogéneo, de acceso directo y dimensión fija.

vector
Tabla unidimensional.

Bibliografía

Botella, P.; Bofill, M.; Burgués, X.; Franch, X.; Lagonigro, R; Vancells, J. (1998). *Fundamentos de programación I*. Barcelona: Ediuoc.

Castro, J.; Cucker, F.; Messeguer, X.; Rubio, A.; Solano, L.; Valles, B. (1992). *Curs de programació*. Madrid, etc.: McGraw-Hill.