Introducción a la Programación usando Pascal como primer lenguaje

Capítulo 6
Tipos escalares y tuplas

Francisco J. Ballesteros

Tipos escalares y tuplas

6.1. Otros mundos

En los programas que hemos realizado hemos utilizado **tipos de datos primitivos** de Pascal. Esto es, tipos que forman parte del mundo según lo ve Pascal: **`integer`**, **`real`**, **`char`** y **`boolean`**.

Pero hemos tenido problemas de claridad y de falta de abstracción a la hora de escribir programas que manipulen entidades que no son ni enteros, ni reales, ni caracteres, ni valores de verdad. Por ejemplo, si queremos manipular fechas vamos a necesitar manipular meses y también días de la semana. Si hacemos un programa para jugar a las cartas vamos a tener que entender lo que es Sota, Caballo y Rey. Hasta el momento hemos tenido que utilizar enteros para ello. En realidad nos da igual si el ordenador utiliza siempre enteros para estas cosas. Lo que no queremos es tenerlo que hacer nosotros.

Utilizar enteros y otros tipos conocidos por Pascal para implementar programas que manipulan entidades abstractas (que no existen en el lenguaje) es un error. Los programas resultarán complicados rápidamente y pronto no sabremos lo que estamos haciendo. Cuando veamos algo como

```
if c = 8 then begin
    ...
end;
```

no sabremos si **`c`** es una carta y "**`8`**'' es en realidad una **`Sota`** o qué estamos haciendo (a esto ayuda el críptico nombre "**`c`**'' elegido para este ejemplo). Tendremos problemas serios para entender nuestro propio código y nos pasaremos mucho tiempo localizando errores y depurándolos.

Pascal permite **definir nuevos tipos** para inventarnos mundos que no existen inicialmente en Pascal (ver figura 1).

Como ya sabemos un tipo de datos no es otra cosa que un conjunto de elementos identificado por un nombre, con una serie de operaciones y al que pertenecen una serie de entidades homogéneas entre sí.

Pues bien, una forma de inventarse un nuevo mundo (al que pertenecerán entidades que no pueden mezclarse con las de otros mundos) es declarar un nuevo tipo de datos. Una de las formas que tenemos para hacer esto es **enumerar** los elementos del nuevo tipo de datos. A los tipos definidos así se los conoce como **tipos enumerados**. En un tipo enumerado los elementos (los literales) son enumerables y se le puede asignar un número natural a cada elemento.

En Pascal, una declaracion de un nuevo tipo de datos define dicho tipo y le da

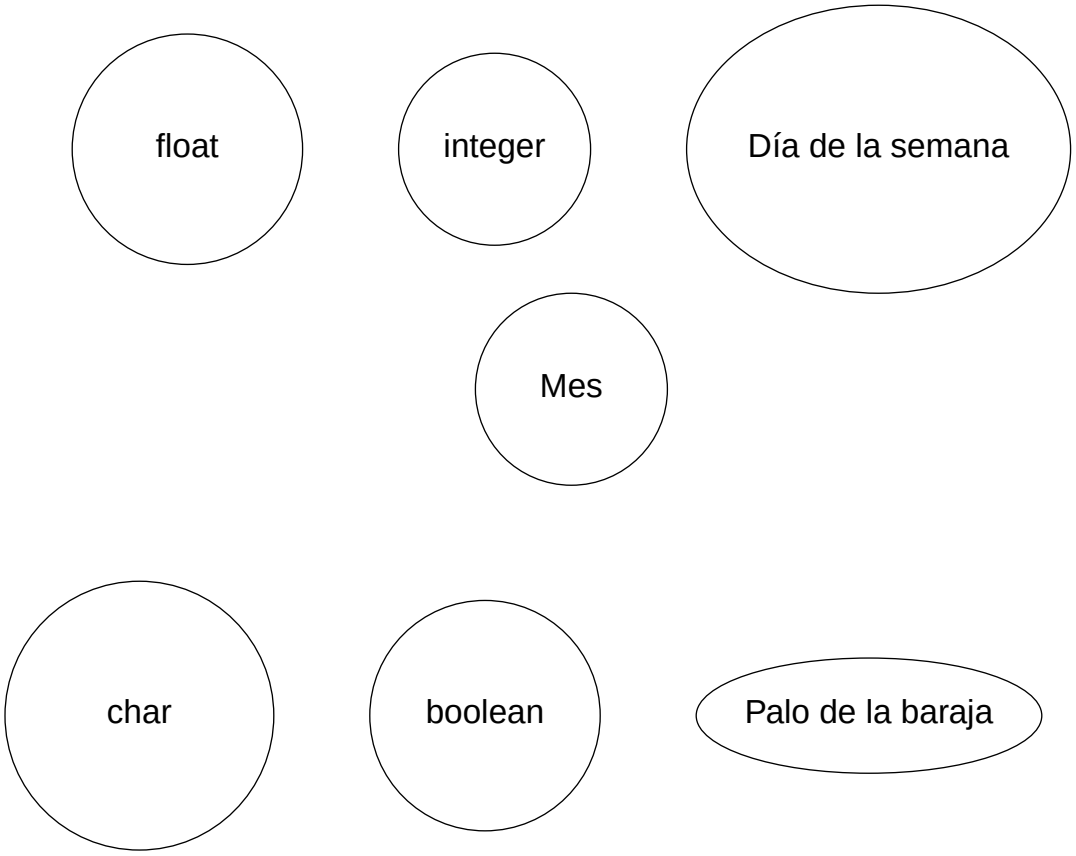

Fig. 1. Además de enteros , caracteres, reales y booleanos podemos inventar nuevos mundos.

un nombre. Las declaraciones de tipos deben estar en una sección que comienze por la palabra reservada "**type**'' como por ejemplo:

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

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

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

Naturalmente podríamos también definir:

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

O tal vez queramos manipular dígitos de números romanos:

```
TipoDigitoRomano = (I, V, X, L, C, D, M);
```

Como puede verse, inventarse otro tipo es fácil para tipos enumerados. Basta enumerar con esta sintaxis los elementos del conjunto. Tras cada una de estas declaraciones disponemos de un nuevo tipo de datos. Por ejemplo, tras la primera declaración mostrada disponemos del nuevo identificador **`TipoDiaSem`** que corresponde con el tipo de datos para días de la semana; tras la segunda, **`TipoMes`**; etc.

Todos los nombres que hemos escrito entre paréntesis a la hora de enumerar los elementos del nuevo tipo son, a partir del momento de la declaración del tipo, nuevos literales (valores constantes) que podemos utilizar en el lenguaje. Naturalmente, son literales del tipo en el que los hemos enumerado. Así por ejemplo, **`Lun`** y **`Vie`** son a partir de ahora literales del tipo **`TipoDiaSem`**.

Para el lenguaje, **`Vie`** no es muy diferente de "**`3`**'' o cualquier otro literal. La diferencia radica en que el primero es de tipo **`TipoDiaSem`** y el segundo de tipo **`integer`**. De hecho, **`Vie`** estará representado dentro del ordenador por un número entero, por ejemplo el 4 o el 5. Pero eso no quiere decir que sea un **`integer`**.

Esto quiere decir que ahora podríamos, por ejemplo, declarar una variable del tipo **`TipoDiaSem`** como sigue:

```
var
    dia: TipoDiaSem;
```

O una constante, como hacemos a continuación:

```
const
    UnBuenDia: TipoDiaSem = Vie;
```

Si es que consideramos el viernes como una constante universal para un buen día.

Es importante utilizar siempre el mismo estilo para los identificadores de nuevos tipos. Principalmente, resulta útil distinguirlos de variables, funciones y otros artefactos tan sólo con ver sus nombres. En este curso los nombres de tipo deberían siempre comenzar por "**`Tipo`**'' y capitalizarse de forma similar a como vemos en los ejemplos. Los identificadores para los elementos del tipo también estarán capitalizados del mismo modo, con cada inicial en mayúscula. Queremos nombres fáciles de escribir y, sobre todo, fáciles de leer. Estos nombres deberían ser nombre cortos y claros, pero eso sí, es bueno que no sean nombres populares para nombrar variables o funciones. (No deberían coincidir con nombres de procedimiento, que deberían corresponder a las acciones que realizan). Por ejemplo, los literales empleados anteriormente para definir un dígito romano son pésimos. Posiblemente fuese mejor utilizar algo como

```
TipoDigitoRom = (RomI, RomV, RomX, RomL, RomC, RomD, RomM);
```

¡En realidad ya conocíamos los tipos enumerados! Los enteros, los caracteres y los booleanos son tipos enumerados (podemos contarlos y sus literales están definidos por un único valor discreto). La principal diferencia entre estos tipos y los nuevos es que no tenemos que definirlos (los que conocíamos ya están predefinidos en

el lenguaje). Así pues, debería resultar sencillo manipular valores y variables de los nuevos tipos enumerados al escribir nuestro programa: lo haremos exactamente igual que hemos hecho con cualquier tipo enumerado de los predefinidos o primitivos del lenguaje.

Por ejemplo, sabemos que este programa lee un carácter, calcula el siguiente carácter y lo imprime (Suponiendo que el carácter leído no es el último):

```
 1 program siguiente;
 2 var
 3     c: char;
 4 begin
 5     readln(c);
 6     c := succ(c);
 7     writeln(c);
 8 end.
```

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

```
 1 program sigdia;
 2
 3 type
 4     TipoDiaSem = (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
 5 var
 6     dia: TipoDiaSem;
 7 begin
 8     readln(dia);
 9     dia := succ(dia);
10     writeln(dia);
11 end.
```

Igual que deberíamos haber hecho con el programa para el siguiente carácter, haríamos bien en no intentar calcular el siguiente día al domingo: esto es un error y el programa se detendría. Si ejecutamos nuestro programa y escribimos "**Dom**" en la entrada...

```
unix% fpc sigdia.p
unix% digdia
unix% Dom
Runtime error 107 at 00010E9F
unix%
```

Según el manual de nuestro compilador, el error 107 corresponde a un error de

valor fuera de rango. Hemos intentado que una variable adopte un valor que no existe para su tipo de datos.

Podemos por ejemplo considerar que a un domingo le sigue un lunes y cambiar el programa según se ve a continuación:

```
 1 {
 2     Siguiente dia al leido
 3 }
 4 program sigdia;
 5
 6 type
 7     TipoDiaSem = (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
 8
 9 function sigdia(d: TipoDiaSem): TipoDiaSem;
10 begin
11     if d = Dom then begin
12         result := Lun;
13     end
14     else begin
15         result := succ(d);
16     end;
17 end;
18
19 var
20     dia: TipoDiaSem;
21 begin
22     readln(dia);
23     writeln(sigdia(dia));
24 end.
```

Como puede verse, podemos declarar, inicializar, asignar y consultar variables de un tipo enumerado del mismo modo que si fuesen variables de cualquier otro tipo. Además, los operadores de comparación también está definidos. El orden de los literales del tipo enumerado es precisamente el orden utilizado en su declaración (de menor a mayor).

Las funciones predefinidas de Pascal llamadas **`succ`** y **`pred`** aceptan cualquier tipo enumerado. En general, todo lo que podemos hacer con **`char`** también lo podemos hacer con un tipo enumerado (los enteros y los booleanos son un poco especiales puesto que tienen operaciones exclusivas de ellos, como las aritméticas para los enteros y las del álgebra de bool para los booleanos).

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

`TipoDiaSemana`:

`posdom := ord(Dom);`

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

`read(dia);`

y

`write(dia);`

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

Existen dos funciones predefinidas, "**`high`**" y "**`low`**" que devuelven en menor y mayor valor del tipo enumerado que se indica como argumento y resultan útiles para detectar límites. Por ejemplo, nuestro programa podría haberse escrito como sigue.

```
 1
 2 {
 3     Siguiente dia al leido
 4 }
 5 program sigdia;
 6
 7 type
 8     TipoDiaSem = (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
 9
10 function sigdia(d: TipoDiaSem): TipoDiaSem;
11 begin
12     if d = high(TipoDiaSem) then begin
13         result := low(TipoDiaSem);
14     end
15     else begin
16         result := succ(d);
17     end;
18 end;
19
20 var
21     dia: TipoDiaSem;
22 begin
23     readln(dia);
24     writeln(sigdia(dia));
25 end.
```

6.2. Mundos paralelos y tipos universales

En Pascal podemos definir tipos a partir de otros tipos. Cuando hacemos esto, una

variable del nuevo tipo definido puede adquirir cualquiera de los valores disponibles para el tipo original, y además **es compatible** con el tipo original. En otros lenguajes, esta misma definición daría lugar a un tipo **incompatible** con el original, así que consulta el manual de tu lenguaje.

Por ejemplo, en:

```
type
    TipoElemento = char;
    TipoManzanas = integer;
    TipoPeras = integer;
```

se han declarado tres nuevos tipos de datos: **`TipoElemento`**, **`TipoManzanas`** y **`TipoPeras`**. El primer tipo tiene los mismos valores que el tipo de datos **`char`**. Y se puede mezclar con los **`char`**. Esto es, **`TipoElemento`** es un **sinónimo** del tipo **`char`** desde que se ha definido.

Los tipos **`TipoManzanas`** y **`TipoPeras`** son similares a **`integer`**. Una variable **`p`** de tipo *TipoPeras* puede tomar como valor un entero cualquiera, lo mismo que **`integer`**. Pasaría lo mismo con una variable **`m`** de tipo **`TipoManzanas`**.

6.3. Subrangos

Hay otra forma de declarar nuevos conjuntos de elementos. Por ejemplo, en la realidad (en matemáticas) tenemos los enteros pero también tenemos los números positivos. Igualmente, tenemos los días de la semana pero también tenemos los días laborables. Los positivos son un subconjunto de los enteros y los días laborables son un subconjunto de los días de la semana. En Pascal tenemos **subrangos** para expresar este concepto.

Así, un día laborable es en realidad un día de la semana pero ha de ser un día comprendido en el intervalo de lunes a viernes. Esto lo podemos expresar en Pascal como sigue:

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

A partir de esta declaración tenemos el nuevo nombre de tipo **`TipoDiaLab`** y podemos escribir

```
var
    diareunion: TipoDiaLab;
```

para declarar, por ejemplo, una variable para el día de una reunión. Dicha variable sólo podrá tomar como valor días comprendidos entre **`Lun`** y **`Vie`** (inclusive ambos). Eso es lo que quiere decir el rango **`Lun..Vie`** en la declaración del subrango: la lista de días comprendidos entre **`Lun`** y **`Vie`** incluyendo ambos extremos del intervalo. Por cierto, al tipo empleado para definir el subrango se le llama **tipo base** del subrango. Por ejemplo, decimos que **`TipoDiaSem`** es el tipo base de **`TipoDiaLab`**. Dicho de otro

modo: los días laborables son días de la semana.

Dado que una variable de tipo **TipoDiaLab** es también del tipo **TipoDiaSem** ambos tipos se consideran compatibles y podemos asignar variables de ambos tipos entre sí (con cuidado de que no se salgan del rango, claro). Por ejemplo, si la variable **hoy** es de tipo **TipoDiaSem**, podríamos hacer esto:

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

Eso sí, si en algún momento intentamos ejecutar algo como

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

el programa sufrirá un error en tiempo de ejecución en cuanto se intente realizar la asignación. Su ejecución se detendrá tras imprimir un mensaje de error informando del problema. ¿Qué otra cosa te gustaría que pasara si te ponen una reunión un domingo?

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

```
type
    TipoNatural = 0..high(Integer);
    TipoDiaDelMes = 1..31;
```

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

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

```
type
    TipoLetraMay = 'A'..'Z';
    TipoLetraMin = 'a'..'z';
    TipoDigito = '0'..'9';
```

En realidad ya hemos conocido los subrangos. Cuando hablamos de la sentencia *case* vimos que una de las posibilidades a la hora de declarar un conjunto de valores era utilizar un rango, como en

```
case c of
'A'..'Z':
    ...
```

```
'0'..'9':
    ...
end;
```

6.4. Registros y tuplas

Los tipos utilizados hasta el momento tienen elementos simples, definidos por un único valor. Se dice que son **tipos elementales** o tipos simples. ¿Pero qué hacemos si nuestro programa ha de manipular objetos que no son simples? Necesitamos también los llamados **tipos compuestos**, construidos a partir de los tipos simples.

Por ejemplo, un programa que manipula cartas debe tener probablemente constantes y variables que se refieren a cartas. ¿Qué tipo ha de tener una carta?

Anteriormente realizamos un programa que manipulaba ecuaciones de segundo grado y soluciones para ecuaciones de segundo grado. Terminamos declarando cabeceras de procedimiento como

```
procedure solucionarec(a, b, c: real; var x1, x2: real);
```

cuando en realidad habría sido mucho más apropiado poder declarar:

```
procedure solucionarec(ec: TipoEcuacion; varf sol: TipoSolucion)
```

o incluso

```
function resolver(ec: TipoEcuacion): TipoSolucion;
```

Justo en ese programa, las declaraciones de variables del programa principal eran estas:

```
var
    a, b, c: real;    { ax2+xb+c=0 }
    x1, x2: real;
```

Pero en realidad habríamos querido declarar esto:

```
var
    ec: TipoEcuacion;
    sol: TipoSolucion;
```

¿Qué es una carta? ¿Qué es una ecuación de segundo grado? ¿Y una solución para dicha ecuación? Una carta es un palo y un valor. Una ecuación son los coeficientes$a$, b y c. Una solución son dos v alores, digamos x_1 y x_2. Todas estas cosas son **tuplas**, o elementos pertenecientes a un producto cartesiano.

El concepto de producto cartesiano define un nuevo conjunto de elementos a partir de conjuntos ya conocidos. Por ejemplo, una carta es en realidad un par (*valor* , *palo*) y forma parte del producto cartesiano *TipoValor* x *TipoPalo*.

En Pascal y otros muchos lenguajes es posible definir estos productos cartesianos declarando tipos de datos conocidos como **registros** o **records**.

Vamos a ver algunos ejemplos. Aunque sería más apropiado utilizar un nuevo

tipo enumerado para el caso de la baraja española, supondremos que nuestra baraja tiene las cartas numeradas de uno a doce. Luego podemos definir un nuevo tipo como sigue:

```
type
    TipoValor = 1..12;
```

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

```
type
     TipoPalo = (Oros, Bastos, Copas, Espadas);
```

Ahora podemos definir un tipo nuevo para una pareja de ordenada compuesta por un valor y un palo. La idea es la misma que cuando construimos tuplas con un producto cartesiano.

```
type
    TipoCarta = record
        palo: TipoPalo;
        valor: TipoValor;
    end;
```

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

Luego algo de tipo **`TipoCarta`** es en realidad una tupla formada por algo de tipo **`TipoValor`** y algo de tipo **`TipoPalo.`** Eso es razonable. El “2 de bastos’’ es en realidad algo definido por el “2’’ y por los “bastos’’.

Para una ecuación de segundo grado podríamos definir:

```
type
    { a * x**2 + b * x + c = 0 }
    TipoEc = record
        a, b, c: real;
    end;
```

En este caso una ecuación es una terna ordenada de tres cosas (todas de tipo **`real`**). La primera es algo llamado **`a`**, la segunda **`b`** y la tercera **`c`**.

Cada elemento del *record* se denomina **campo** del *record*. Así, una carta será un *record* con dos campos: un campo denominado **`valor`** y otro campo denominado **`palo`**. Es importante saber que estos campos mantienen siempre el mismo orden. En una carta tendremos primero el valor y después el palo.

En la memoria del ordenador una variable (o un valor) de tipo **`TipoCarta`** tiene el aspecto de dos variables guardadas una tras otra: una primero de tipo **`TipoValor`** y otra después de tipo **`TipoPalo`**. La figura 2 muestra el aspecto de dos *records* en la memoria del ordenador.

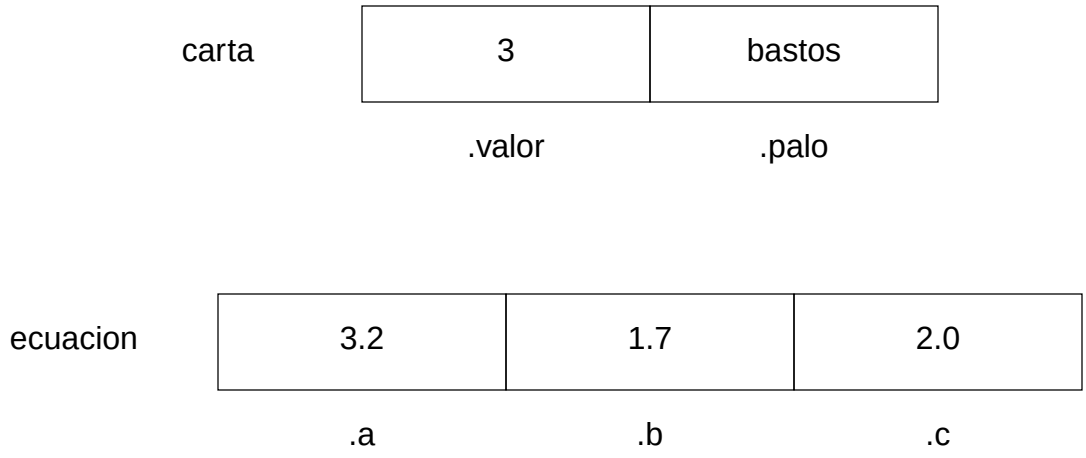

Fig. 2. Aspecto de los records de los ejemplos en la memoria del ordenador.

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

```
var
    carta1: TipoCarta;
    carta2: TipoCarta;
begin
        ...
    carta1 := carta2;    {asigna el valor de carta2 a carta1}
```

No obstante, hay diferencias entre los registros y el resto de los tipos que hemos visto hasta el momento en cuanto a qué operaciones se pueden hacer con ellos:

- Es posible utilizar *records* como parámetros de funciones y procedimientos y como objetos devueltos por funciones. En particular, es posible asignarlos entre sí (se asignan los campos uno por uno, como cabría esperar).
- **No** es posible comparar *records* (del mismo tipo).
- **No** es posible ni leer ni escribir *records*. Para leer registros tenemos que leer campo por campo y para escribirlos tenemos que escribir campo por campo.

Necesitamos poder referirnos a los campos de un *record* de forma individual. Bien para consultar su valor, bien para actualizarlo. Eso se puede hacer con la llamada **notación punto**. Para referirse al campo de un *record* basta utilizar el nombre del *record* (de la variable o constante de ese tipo) y escribir a continuación un "**`.`**'' y el nombre del campo que nos interesa. Por ejemplo, considerando **`carta1`**, de **`TipoCarta`**, podríamos decir:

```
carta1.valor := 1;
carta1.palo := Bastos;
```

Esto haría que **carta1** fuese el as de bastos. La primera sentencia asigna el valor **1** al campo **valor** de **carta1**.

Podemos utilizar **carta1.valor** en cualquier sitio donde podamos utilizar una variable de tipo **TipoValor**. Igual sucede con cualquier otro campo. Un campo se comporta como una variable del tipo al que pertenece (el campo).

Recuerda que un *record* es simplemente un tipo de datos que agrupa en una sola cosa varios elementos, cada uno de su propio tipo de datos. En la memoria del ordenador una variable de tipo *record* es muy similar a tener juntas varias variables (una por cada campo, como muestra la figura 2). No obstante, a la hora de programar la diferencia es abismal. Es infinitamente más fácil manipular cartas que manipular parejas de variables que no tienen nada que ver entre sí. Para objetos más complicados la diferencia es aún mayor.

Al declarar constantes de tipo *record* es útil la sintaxis que tiene el lenguaje para expresar **agregados**. Un agregado es simplemente una secuencia ordenada de elementos que se consideran agregados para formar un elemento más complejo. Esto se verá claro con un ejemplo; estos son el as de bastos y el rey de bastos:

```
const
    AsBastos: TipoCarta = (
        palo: Bastos;
        valor: 1;
    );
    ReyBastos: TipoCarta = (
        palo: Bastos;
        valor: 12;
    );
```

Como puede verse, hemos inicializado la constante escribiendo el nombre del tipo y un "**=**'' tras el que escribimos entre paréntesis, una definición para cada campo del registro, indicando el nombre y valor de cada campo.

Veamos otro ejemplo para seguir familiarizándonos con los *records*. Este procedimiento lee una carta. Como tiene efectos laterales, al leer de la entrada, no podemos implementarlo con una función.

```
procedure leercarta(var c: TipoCarta);
begin
    read(c.valor);
    read(c.palo);
end;
```

Dado que ambos campos son de tipos elementales (no son de tipos compuestos de varios elementos, como los records) podemos leerlos directamente.

6.5. Abstracciones

Es muy útil podernos olvidar de lo que tiene dentro un *record*; del mismo modo que resulta muy útil podernos olvidar de lo que tiene dentro un subprograma.

Si hacemos un programa para jugar a las cartas, gran parte del programa estará manipulando variables de tipo carta sin prestar atención a lo que tienen dentro. Por ejemplo, repartiéndolas, jugando con ellas, pasándolas de la mano del jugador a la mesa, etc. Igualmente, el programa estará continuamente utilizando procedimientos para repartirlas, para jugarlas, para pasarlas a la mesa, etc. Si conseguimos hacer el programa de este modo, evitaremos tener que considerar todos los detalles del programa al programarlo. *Esta es la clave para hacer programas: abstraer y olvidar los detalles*.

Los *records* nos permiten abstraer los datos que manipulamos (verlos como elementos abstractos en lugar de ver sus detalles) lo mismo que los subprogramas nos permiten abstraer los algoritmos que empleamos (verlos como operaciones con un nombre en lugar de tener que pensar en los detalles del algoritmo). Si tenemos que tener en la mente todos los detalles de todos los algoritmos y datos que empleamos, ¡Nunca haremos ningún programa decente que funcione! Al menos ninguno que sea mucho más útil que nuestro “hola π”. Si abstraemos... ¡Seremos capaces de hacer programas tan elaborados como sea preciso!

Igual que las construcciones como el *if* (y otras que veremos) permiten estructurar el código del programa (el flujo de control), con las construcciones como *record* (y otras que veremos) podemos estructurar los datos. Por eso normalmente se habla de **estructuras de datos** en lugar de hablar simplemente de datos. Igualmente se habla de **estructuras de control** para referirse a sentencias como *if*, *case* y otras que estructuran el flujo de control.

Los datos están estructurados en tipos de datos complejos hechos a partir de tipos de datos más simples; hechos a su vez de tipos mas simples... hasta llegar a los tipos básicos del lenguaje. Podemos tener *records* cuyos campos sean otros *records*, etc. Estructurar los datos utilizando tipos de datos compuestos de tipos más simples es similar a estructurar el código de un programa utilizando procedimientos y funciones.

Recuerda lo que dijimos hace tiempo: *Algoritmos + Estructuras = Programas.*

6.6. Geometría

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

```
type
```

```
TipoPunto = record
    x,y: integer;
end;
```

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

```
const
    Origen: TipoPunto = (x: 0; y: 0);
```

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

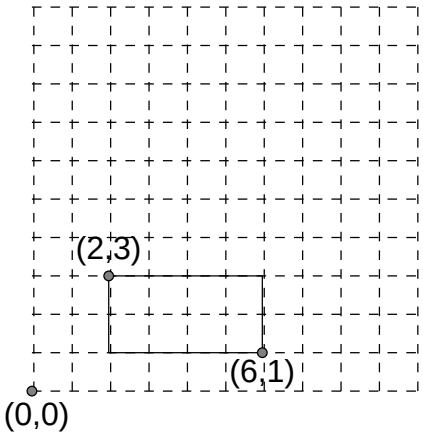

Fig. 3. Coordenadas en dos dimensiones incluyendo origen y un rectángulo.

Si ahora queremos representar un rectángulo, podríamos hacerlo definiendo el punto de arriba a la izquierda y el punto de abajo a la derecha en el rectángulo, como muestra la figura 3. Podríamos llamar a ambos puntos el menor y el mayor punto del rectángulo y declarar un nuevo tipo como sigue:

```
TipoRect = record
    min, max: TipoPunto;
end;
```

Ahora, dado un rectángulo

```
r: TipoRect;
```

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

Un círculo podríamos definirlo a partir de su centro y su radio. Luego podríamos declarar algo como:

```
TipoCirculo = record
    centro: TipoPunto;
    radio: integer;
end;
```

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

```
TipoElipse = record
    centro: TipoPunto;
    radiomin, radiomax: integer;
end;
```

Podemos centrar una elipse en un círculo haciendo algo como

```
elipse.centro := circulo.centro;
```

lo que modifica el centro de la elipse (que es de **`TipoPunto`**) para que tenga el mismo valor que el del centro del círculo.

Vamos ahora a inscribir un círculo en un cuadrado tal y como muestra la figura 4. Necesitamos definir tanto el centro del círculo como su radio:

```
circulo.centro.x := (cuadrado.min.x + cuadrado.max.x) div 2;
circulo.centro.y := (cuadrado.min.y + cuadrado.max.y) div 2;
circulo.radio := (cuadrado.max.x - cuadrado.min.x) div 2;
```

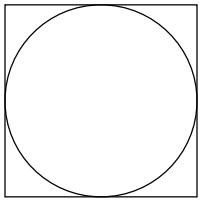

Fig. 4. Círculo inscrito en un cuadrado.

Hemos supuesto que nuestro *cuadrado* es en realidad un rectángulo. ¿Hemos perdido el juicio? Puede ser, pero es que nuestra variable *cuadrado* es de tipo *TipoRectangulo*. Por supuesto, dado su nombre, debería ser un cuadrado. ¿Cómo sabemos si es un cuadrado? Con esta expresión de tipo **`boolean`**:

```
cuadrado.max.x - cuadrado.min.x = cuadrado.max.y - cuadrado.min.y
```

Confiamos en que la mecánica de declaración de records y el uso elemental de los mismos, para operar con ellos y para operar con sus campos, se entienda mejor tras ver

estos ejemplos. Ahora vamos a hacer programas con todo lo que hemos visto.

6.7. Aritmética compleja

Queremos sumar, restar multiplicar y dividir números complejos. Para empezar, podríamos definir un tipo de datos para un número complejo (en coordenadas cartesianas):

```
type
    TipoCompl = record
        re, im: real;
    end;
```

Antes siquiera de pensar en cuál es el subproblema más pequeño por el que podemos empezar, considerando que tenemos un nuevo tipo de datos, parece que podríamos definir una función para construir un nuevo valor de tipo **`TipoCompl`**, un procedimiento para leer un número complejo de la entrada y otro para escribir un número complejo en la salida. Nuestro programa quedaría como sigue. Nótese el tipo de paso de parámetros que hemos empleado en cada ocasión. Sólo el procedimiento **`leercompl`** emplea paso de parámetros por referencia.

```
 1
 2 {
 3     Complejos
 4
 5 }
 6 program compl;
 7
 8 type
 9     TipoCompl = record
10         re, im: real;
11     end;
12
13 function nuevocompl(re, im: real): TipoCompl;
14 var c: TipoCompl;
15 begin
16     c.re := re;
17     c.im := im;
18     result := c;
19 end;
20
21 procedure leercompl(var c: TipoCompl);
22 begin
23     read(c.re);
24     read(c.im);
25 end;
26
27 procedure escrcompl(c: TipoCompl);
```

```
28 begin
29     if c.im > 0.0 then begin
30         write(c.re:0:2, '+', c.im:0:2, 'i');
31     end
32     else if c.im = 0.0 then begin
33         write(c.re:0:2);
34     end
35     else begin
36         write(c.re:0:2, '-', c.im:0:2, 'i');
37     end;
38 
39 end;
40 
41 procedure escrcomplln(c: TipoCompl);
42 begin
43     escrcompl(c);
44     writeln();
45 end;
46 
47 var
48     c: TipoCompl;
49 begin
50     leercompl(c);
51     escrcomplln(c);
52 end.
```

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

```
unix% fpc compl.p
unix% compl
unix% 30 8
30.00+8.00i
```

Hemos de recordar que en Pascal podríamos escribir agregados para constantes de tipo complejo. Pero no queremos saber nada de los detalles respecto a cómo está hecho un número complejo. Por eso hemos definido una función llamada **nuevocompl** para crear un número complejo. Nos gustaría que nuestro programa manipulase los números complejos utilizando las operaciones que definamos, del mismo modo que hacemos con los enteros.

Sumar y restar números complejos es fácil: sumamos y restamos las partes reales e imaginarias. Ambas operaciones podrían ser funciones que devuelvan nuevos valores complejos con el resultado de la operación.

```
function sumacompl(c1, c2: TipoCompl): TipoCompl;
begin
```

```
        c1.re := c1.re + c2.re;
        c1.im := c1.im + c2.im;
        result := c1;
    end;

    function restacompl(c1, c2: TipoCompl): TipoCompl;
    begin
        c1.re := c1.re - c2.re;
        c1.im := c1.im - c2.im;
        result := c1;
    end;
```

Podríamos continuar así definiendo funciones para la multiplicación, división, conjugado, etc. Una vez programadas estas operaciones cualquier programa que manipule números complejos puede olvidarse por completo de cómo están hechos éstos. Y si descubrimos que en un programa determinado necesitamos una operación que se nos había olvidado, la definimos. Por ejemplo, es muy posible que necesitemos funciones para obtener la parte real y la parte imaginaria de un número complejo. Una alternativa sería utilizar **`c.re`** y **`c.im`**, lo que es más sencillo, pero rompe la abstracción puesto que vemos cómo está hecho un número complejo.

Este programa suma un complejo que lee con su conjugado.

```
 1
 2 {
 3     Complejos
 4
 5 }
 6 program compl;
 7
 8 type
 9     TipoCompl = record
10         re, im: real;
11     end;
12
13 const
14     CeroCompl: TipoCompl = (
15         re: 0.0;
16         im: 0.0;
17     );
18
19 function nuevocompl(re, im: real): TipoCompl;
20 var
21     c: TipoCompl;
22 begin
23     c.re := re;
```

```
24      c.im := im;
25      result := c;
26  end;
27
28  procedure leercompl(var c: TipoCompl);
29  begin
30      read(c.re);
31      read(c.im);
32  end;
33
34  procedure escrcompl(c: TipoCompl);
35  begin
36      if c.im > 0.0 then begin
37          write(c.re:0:2, '+', c.im:0:2, 'i');
38      end
39      else if c.im = 0.0 then begin
40          write(c.re:0:2);
41      end
42      else begin
43          write(c.re:0:2, '-', c.im:0:2, 'i');
44      end;
45
46  end;
47
48  procedure escrcomplln(c: TipoCompl);
49  begin
50      escrcompl(c);
51      writeln();
52  end;
53
54  function conjugado(c: TipoCompl): TipoCompl;
55  begin
56      c.im := - c.im;
57      result := c;
58  end;
59
60
61  function sumacompl(c1, c2: TipoCompl): TipoCompl;
62  begin
63      c1.re := c1.re + c2.re;
64      c1.im := c1.im + c2.im;
65      result := c1;
66  end;
67
68  var
69      c: TipoCompl;
70  begin
71      leercompl(c);
```

```
72      escrcomplln(sumacompl(c, conjugado(c)));
73  end.
```

6.8. Cartas del juego de las 7 y 1/2

Queremos hacer un programa para jugar a las 7½ y, por el momento, queremos un programa que lea una carta de la entrada estándar y escriba su valor según este juego. Este es un juego para la baraja española donde las figuras valen ½ y el resto de cartas su valor nominal.

Vamos a hacer con los datos lo mismo que hemos hecho con los subprogramas: ¡Suponer que ya los tenemos! En cuanto sean necesarios, eso sí. Por ejemplo, parece que claro que lo debemos hacer es:

1. Leer una carta
2. Calcular su valor
3. Imprimir su valor.

Pues entonces... ¡Hacemos justo eso!

```
program sieteymedia;
var
    carta: TipoCarta;
    valor: real;
begin
    leercarta(carta);
    valor := valorcarta(c);
    writeln('puntos = ', valor:0:2);
end.
```

Dado que un valor es un número real en este caso (hay cartas que valen ½) sabemos que **`valor`** debe ser un **`real`** y que podemos utilizar **`writeln`** para escribir el valor.

Hemos hecho casi lo mismo con la carta. ¡Aunque Pascal no tiene ni idea de lo que es una carta! Nos hemos inventado el tipo **`TipoCarta`**, que después tendremos que definir en la sección correspondiente, y hemos declarado una variable de ese tipo. Como queremos leer una carta, nos hemos inventado **`leercarta`** (que necesita que le demos la variable cuyo valor hay que leer). Igualmente, **`valorcarta`** es una función que también nos hemos sacado de la manga para que nos devuelva el valor de una carta. ¡Fácil! ¿no?

Antes de escribir los procedimientos, resulta muy útil definir el tipo de datos (muchas veces el tipo que empleemos determinará que aspecto tiene la implementación de los procedimientos y funciones que lo manipulen). Una carta es un palo y

un valor. Pues nos inventamos dos tipos, *TipoPalo* y *TipoValor*, y declaramos:

```
TipoCarta = record
    palo: TipoPalo;
    valor: TipoValor;
end;
```

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

```
TipoPalo = (Oros, Bastos, Copas, Espadas);
TipoValor = 1..12;
```

Así va surgiendo el programa. Un procedimiento para leer una carta necesitará leer el palo y leer el valor; y tendrá un parámetro pasado por referencia para la carta.

```
procedure leercarta(var c: TipoCarta);
begin
    read(c.valor);
    read(c.palo);
end;
```

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

¿Cómo implementamos el subprograma que nos da el valor de una carta? Claramente es una función: le damos una carta y nos devuelve su valor. Bueno, todo depende de si la carta es una figura o no. Si lo es, entonces su valor es 0.5; en otro caso su valor es el valor del número de la carta. Podemos entonces escribir esto por el momento...

```
function valorcarta(c: TipoCarta): real;
begin
    if esfigura(c) then begin
        result := 0.5;
    end
    else begin
        result := c.valor
    end;
end;
```

De nuevo, nos hemos inventado **esfigura**. Se supone que esa función devolverá **True** cuando tengamos una carta que sea una figura:

```
function esfigura(c: TipoCarta): boolean;
begin
    result := c.valor >= 10;
```

```
    end;
```

Recuerda que lo que es (o no es) una figura es la carta, no el valor de la carta. Por eso nuestra función trabaja con una carta y no con un valor.

Fíjate en que todo el tiempo tratamos de conseguir que el programa manipule cartas u otros objetos que tengan sentido en el problema en que estamos trabajando. Si se hace esto así, la estructura de los datos y la estructura del programa saldrá sola. Es curioso que algunos escultores afirmen que lo mismo sucede con las esculturas, que estaban ya dentro de la piedra y que ellos sólo tenían que retirar lo que sobraba.

Y ya está. El programa está terminado.

```
 1
 2 {
 3     Jugar a las 7 y 1/2
 4 }
 5 program sieteymedia;
 6
 7 type
 8     TipoPalo = (Oros, Bastos, Copas, Espadas);
 9     TipoValor = 1..12;
10     TipoCarta = record
11         palo: TipoPalo;
12         valor: TipoValor;
13     end;
14
15
16 procedure leercarta(var c: TipoCarta);
17 begin
18     read(c.valor);
19     read(c.palo);
20 end;
21
22 procedure escrcarta(c: TipoCarta);
23 begin
24     writeln(c.valor, ' de ', c.palo);
25 end;
26
27 function esfigura(c: TipoCarta): boolean;
28 begin
29     result := c.valor >= 10;
30 end;
31
32 function valorcarta(c: TipoCarta): real;
33 begin
```

```
34      if esfigura(c) then begin
35          result := 0.5;
36      end
37      else begin
38          result := c.valor
39      end;
40  end;
41
42  var
43      c: TipoCarta;
44      valor: real;
45  begin
46      leercarta(c);
47      valor := valorcarta(c);
48      writeln('puntos = ', valor:0:2);
49  end.
```

Aunque no lo hemos mencionado, hemos compilado y ejecutado el programa en cada paso de los que hemos descrito. Si es preciso durante un tiempo tener

```
type
    TipoCarta = integer;
```

para poder declarar cartas, pues así lo hacemos. O, si necesitamos escribir un

```
result := 0;
```

en una función para no escribir su cuerpo, pues también lo hacemos. Lo que importa es poder ir probando el programa poco a poco.

Si el problema hubiese sido mucho más complicado. Por ejemplo, “jugar a las 7½’’, entonces habríamos simplificado el problema todo lo posible, para no abordarlo todo al mismo tiempo. Muy posiblemente, dado que sabemos que tenemos que manipular cartas, habríamos empezado por implementar el programa que hemos mostrado. Podríamos después complicarlo para que sepa manejar una mano de cartas y continuar de este modo hasta tenerlo implementado por completo. La clave es que en cada paso del proceso sólo queremos tener un trozo de código (o datos) nuevo en el que pensar.

6.9. Problemas

Obviamente, en todos los problemas se exige utilizar tipos enumerados y registros allí donde sea apropiado (en lugar de hacerlos como hemos hecho en capítulos anteriores). Como de costumbre hay enunciados para problemas cuya solución ya tienes; deberías hacerlos ahora sin mirar la solución.

1. Ver que día de la semana será mañana dado el de hoy.
2. Calcular los días del mes, dado el mes.
3. Calcular el día del año desde comienzo de año dado el mes y el día del mes.
4. Calcular el día de la semana, suponiendo que el día uno fue Martes, dado el número de un día.
5. Leer y escribir el valor de una carta utilizando también un tipo enumerado para el valor, para que podamos tener sota, caballo, rey y as.
6. Dada una fecha, y suponiendo que no hay años bisiestos y que todos los meses son de 30 días, calcular el número de días desde el 1 enero de 1970.
7. Ver la distancia en días entre dos fechas con las mismas suposiciones que en el problema anterior.
8. Sumar, restar y multiplicar números complejos.
9. Ver si un punto está dentro de un rectángulo en el plano.
10. Mover un rectángulo según un vector expresando como un punto.
11. Ver si una carta tiene un valor mayor que otra en el juego de las 7 y media.
12. Decidir si un punto está dentro de un circulo.
13. Evaluar expresiones aritméticas de dos operandos y un sólo operador para sumas, restas, multiplicaciones y divisiones.
14. Resolver
    los problemas anteriores que tratan de figuras geométricas de tal forma que los datos se obtengan desde la entrada y que puedan darse los puntos en cualquier orden.
15. Decidir se si aprueba una asignatura considerando que hay dos ejercicios, un test y un examen, y que el test se considera apto o no apto y el examen se puntúa de 0 a 10. Para aprobar hay que superar el test y aprobar el examen.
16. Decidir si una nota en la asignatura anterior es mayor que otra.
17. Componer colores primarios (rojo, verde y azul).
18. Ver si un color del arcoiris tiene mayor longitud de onda que otro.
19. Implementar una calculadora de expresiones aritméticas simples. Deben leerse dos operandos y un operador e imprimir el valor de la expresión resultante.
20. Para cada uno de los siguientes objetos, define un tipo de datos en Pascal y haz un procedimiento para construir un nuevo elemento de ese tipo, otro para leer una variable de dicho tipo y otro para escribirla. Naturalmente, debes hacer un programa para probar que el tipo y los procedimientos funcionan. Haz, por ejemplo, que dicho programa escriba dos veces el objeto leído desde la entrada.
    A) Un ordenador, para un programa de gestión de inventario de ordenadores. Un ordenador tiene una CPU dada, una cantidad de memoria instalada dada, una cantidad de disco dado y ejecuta a un número de MHz dado. Tenemos CPUs que pueden ser Core duo, Core 2 duo, Core Quad, Core i7 o Cell. La memoria se mide en números enteros de Mbytes para este problema.

B) Una tarjeta gráfica, que puede ser ATI o Nvidia y tener una CPU que ejecuta a un número de MHz dado (Sí, las tarjetas gráficas son en realidad ordenadores empotrados en una tarjeta) y una cantidad memoria dada expresada en Mbytes.
C) Un ordenador con tarjeta gráfica.
D) Un empleado. Suponiendo que tenemos a Jesús, María y José como empleados inmortales en nuestra empresa y que no vamos a despedir ni a contratar a nadie más. Cada empleado tiene un número de DNI y una edad concreta. Como nuestra empresa es de líneas aéreas nos interesa el peso de cada empleado (puesto que supone combustible).
E) Un empleado con ordenador con tarjeta gráfica.
F) Un ordenador con tarjeta gráfica y programador (suponiendo que hemos reconvertido nuestra empresa de líneas aéreas a una empresa de software).
G) Un ordenador con dos tarjetas gráficas (una para 2D y otra para 3D) cada una de las cuales ha sido montada por uno de nuestros empleados.

21. Muchos de los problemas anteriores de este curso se han resuelto sin utilizar ni tipos enumerados ni registros cuando en realidad deberían haberse programado utilizando éstas facilidades. Localiza dichos problemas y arreglalos.

Índice