Introducción a la Programación usando Pascal como primer lenguaje

Capítulo 4
Problemas de selección

Francisco J. Ballesteros

Problemas de selección

4.1. Secuencias y decisiones

No todos los problemas van a ser problemas que podemos solucionar directamente. Aunque hay muchísimos que lo son y que ya puedes programar en Pascal, tal y como hemos visto en el capítulo anterior.

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

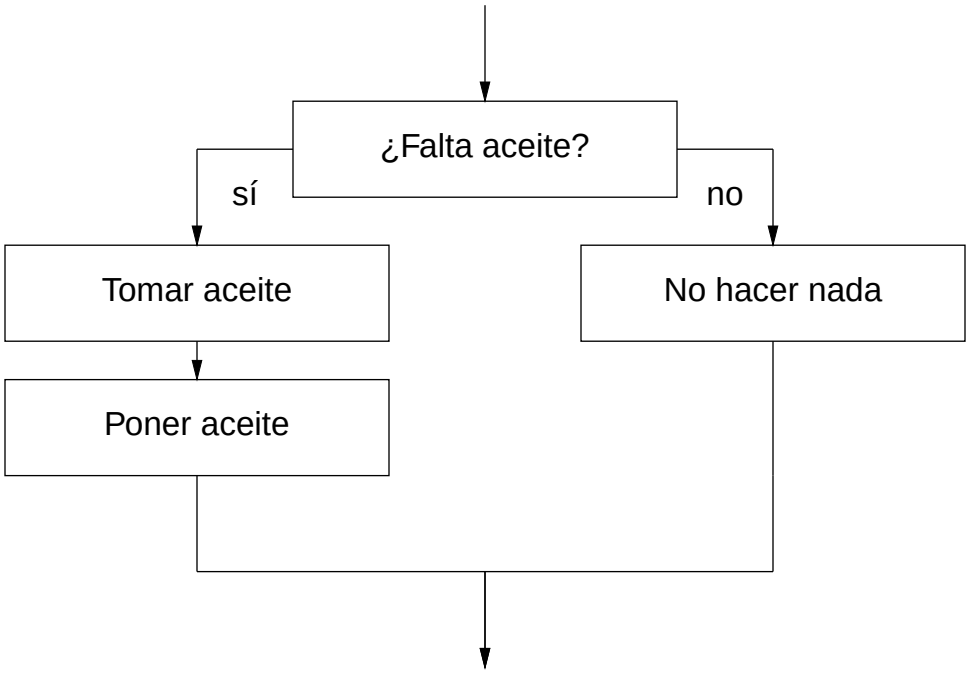

Fig. 1. Decisiones y condiciones

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

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

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

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

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

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

En Pascal tenemos la sentencia (compuesta) llamada comúnmente **if-then-else** (o *si-entonces-sino*) para tomar decisiones y ejecutar unas u otras sentencias en función de que una condición se cumpla (sea cierta) o no (sea falsa). Muchas veces se la llama también **if-then** o simplemente **if** para abreviar.

Por cierto, a esta y otras sentencias que modifican la ejecución secuencial habitual de un programa se las denomina **estructuras de control**. Volviendo al *if*, esta sentencia tiene la forma

```
if condición then          { si ... entonces }
    sentencia
else                       { si no... }
    sentencia
```

Aquí las sentencias que hay dentro pueden ser cualquier cosa, tal como un **writeln** o cualquier otra, incluso otro *if* o cualquier sentencia de las que veremos.

Cuando Pascal tiene que ejecutar este *if-then-else* procede como sigue:

1. Se evalúa la condición del *if*, que ha de ser siempre una expresión de tipo **boolean**. Esto es, una expresión que corresponde a un valor de verdad.
2. Si la condición se cumple (esto es, tiene como valor **True**) entonces se ejecuta la sentencia que hay tras el **then**. A esa sentencia se le llama la "rama del *then*''.
3. Si la condición no se cumple (esto es, tiene como valor **False**) entonces se ejecuta la sentencia que hay tras el **else** (lo que llamamos "rama del *else*'').

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

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

```
function maxint(a, b: integer): integer;
begin
    if a > b then
        result := a
    else
        result := b;
```

```
end;
```

Aquí Pascal evalúa "**a>b**" y en caso que esto sea **True** ejecuta la sentencia

```
result := a
```

En cambio, si la condición es **False** se continúa tras el **else** ejecutando la sentencia

```
result := b
```

A continuación Pascal ejecuta la siguiente sentencia, que en este caso es en realidad el retorno de la función al alcanzar el "**end**" de la misma.

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

Por cierto, debes prestar atención a la **tabulación** o sangrado que se utiliza al escribir sentencias *if-then-else*, como seguramente habrás notado ya.

Las sentencias dentro de la rama *then* están tabuladas más a la derecha; igual sucede con las de la rama *else*. Esto se hace para poner de manifiesto que dichas sentencias están *dentro* de la estructura de control *if-then-else*.

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

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

```
if condición then        { si... entonces }
    sentencia;
```

Como podrás ver aquí sólo hay una rama. Si se cumple la condición se ejecutan la sentencia de esa rama; si no se cumple, se continúa ejecutando el código que tengamos a continuación. Por lo demás, esta sentencia es exactamente igual a la anterior.

Por ejemplo, este código imprime un mensaje para los menores de edad y podría formar parte por ejemplo del cuerpo de un programa principal:

```
if edad < 18 then
    writeln('eres menor de edad');
```

Es posible utilizar un *if-then* dentro de otro, lo que puede dar lugar a confusiones respecto a qué *if* se queda con el *else*.

Veamos este código:

```
program problemas;
const
   N: integer = 10;
begin
   if N < 10 then
       if N = 0 then
           writeln('cero')
   else
       writeln('mayor');
end.
```

¿Qué escribe?

Podría parecer que dada la tabulación, debería escribir **mayor** y termimar. No obstante, el **else** forma parte del último **if** con lo que el programa en realidad (bien tabulado) sería:

```
program problemas;
const
   N: integer = 10;
begin
   if N < 10 then
       if N = 0 then
           writeln('cero')
       else
           writeln('mayor');
end.
```

Y está claro que no entramos al **then** del primer **if** y no escribimos nada.

Para evitar estos problemas, nosotros vamos a utilizar siempre como sentencias dentro de los *if* un bloque de sentencias.

Un **bloque** o **secuencia** es simplemente una o mas sentencias entre un **begin** y un **end**, separadas por "**;**" entre si. Pascal ejecuta las sentencias de un bloque una detrás de otra. Esto es lo que hemos visto en el bloque que utilizamos como cuerpo del programa principal.

Utilizando bloques en la sentencia *if* tendremos código como este:

```
if condición then begin        { si ... entonces }
    sentencia;
    ...
```

```
        sentencia;
    end
    else begin                      { si no... }
        sentencia;
        ...
        sentencia;
    end
```

Tanto la rama *then* como la rama *else* son bloques de sentencias en nuestro caso, y podemos evitar muchos errores si siempre utilizamos bloques de sentencias en este caso incluso cuando el lenguaje de programación no lo requiere.

La función **maxint** quedaría entonces como sigue:

```
function maxint(a, b: integer): integer;
begin
    if a > b then begin
        result := a;
    end
    else begin
        result := b;
    end;
end;
```

Si hacemos lo mismo con el programa que hemos visto mal tabulado, estará más claro lo que está pasando y podemos detectar y arreglar el problema:

```
program problemas;
const
    N: integer = 10;
begin
    if N < 10 then begin
        if N = 0 then begin
            writeln('cero')
        end
        else begin
            writeln('mayor'); { Hmmmm. }
        end;
    end;
end.
```

Si tabulamos mal el **else** veremos que algo anda mal, porque tendremos dos **end** con tabulación que no cuadra y eso nos alerta del error.

```
program problemas;
const
```

```
    N: integer = 10;
begin
    if N < 10 then begin
        if N = 0 then begin
            writeln('cero')
        end
    else begin
        writeln('mayor');
    end;
    end; { Hmmmm. dos end cerrando aquí? Algo anda muy mal... }
end.
```

4.2. Múltiples casos

Para ver otro ejemplo, podemos programar nuestra función **signo** en Pascal como puede verse en el siguiente programa.

```
 1 {
 2     signo
 3 }
 4 program sign;
 5
 6 function signo(n: integer): integer;
 7 begin
 8     if n > 0 then begin
 9         result := 1;
10     end
11     else if n = 0 then begin
12         result := 0;
13     end
14     else begin
15         result := -1;
16     end;
17 end;
18
19 begin
20     writeln(signo(-5));
21     if signo(8) = 1 then begin
22         writeln('8 es positivo');
23     end;
24
25 end.
```

Ignora el programa principal y céntrate en la función **signo** por el momento. En este caso teníamos que distinguir entre tres casos distintos según el número fuese

mayor que cero, cero o menor que cero. Pero el *if-then-else* es binario y sólo sabe distinguir entre dos casos. Lo que hacemos es ir considerando los casos a base tomar decisiones de tipo "sí o no'': si es mayor que cero, entonces el valor es 1; en otro caso ya veremos lo que hacemos. Si pensamos ahora en ese otro caso, puede que **n** sea cero o puede que no. Y no hay más casos posibles. Podríamos haber escrito pues

```
if n > 0 then begin
   result := 1;
end
else begin
   if n = 0 then begin
      result := 0;
   end
   else begin
      result := -1;
   end;
end;
```

A esto se le llama **anidar** un *if-then-else* dentro de otro. En lugar de hacer esto, nosotros hemos encadenado varios *if-then-elses*.

En algunas situaciones es posible que haya que considerar muchos casos, normalmente todos mutuamente excluyentes. Esto es, que nunca podrán cumplirse a la vez las condiciones para dos casos distintos. La función **signo** es un ejemplo de uno de estos casos. En estas situaciones podemos utilizar una variante de la sentencia *if-then-else*, llamada **if-then-else-if** (o **ifs encadenados**) que permite encadenar múltiples *if-then-elses* de una forma más cómoda. Como hemos visto en el ejemplo anterior, esta sentencia tiene el aspecto

```
if condición then begin          { si ... }
    sentencia;
    ...
    sentencia;
end
else if condición then begin                 { y si no, si ...}
    sentencia;
    ...
    sentencia;
end
else if condición then begin                 { y si no, si ...}
    sentencia;
    ...
```

```
    sentencia;
end
else begin                          { y si no ...}
    sentencia;
    ...
    sentencia;
end
```

Esto evita tener que **anidar** *if-then-elses* unos dentro de otros. Como supondrás, podrías quitar el último **`else`** si no lo necesitas.

Cuando utilizamos condicionales para definir el valor de una función, sólo tenemos que tener cuidado de darle un valor en todas las ramas y de que la última rama no tenga condición (sea un **`else`** por ejemplo) para que sea como sea la función tome un valor. Recuerda que si no das un valor a la función, esta aún terminará al llegar a su **`end`** pero devolverá un valor que sea lo que quiera que hubiese en la memoria del ordenador en el sitio en que se espera que esté el resultado de la función.

Los *if-then* se pueden usar como hemos visto para definir funciones en Pascal. Pero se utilizan en general para ejecutar sentencias de forma condicional; cualquier sentencia. Dicho de otro modo: no se trata de poner "**`result:=...`**" rodeados de*if-thens*. Por ejemplo, mira el cuerpo del programa **`problemas`** que hemos visto antes. Otro ejemplo es este código que imprime, en base a la edad de una persona, si es adulto o es menor de edad:

```
if edad < 18 then begin
    writeln('eres menor de edad');
end
else begin
    writeln('eres adulto');
end
```

4.3. Casos

Hay una tercera estructura de control para ejecución condicional de sentencias. Esta estructura se denomina **case** (en otros lenguajes a veces se la llama **switch** por ser esa la palabra reservada en dichos lenguajes para esta sentencia). Para nosotros, la sentencia se llama *case* y está pensada para situaciones en que discriminamos un caso de otro en función de un valor discreto perteneciente a un conjunto (un valor de tipo **`integer`** o de tipo **`char`**, por ejemplo). Su uso está indicado cuando hay múltiples casos excluyentes, de ahí su nombre. La estructura de la sentencia *case* es como sigue:

```
case expresión of
valores:
    sentencia;
```

```
    valores:
        sentencia;
    valores;
        sentencia;
     ...
    otherwise
        sentencia;
    end;
```

Pascal evalúa la expresión primero y luego ejecuta las sentencias de la rama correspondiente al valor que adopta la expresión. Si la expresión no toma como valor ninguno de los valores indicados en un rama, entonces se ejecuta la rama **otherwise**. Por último, como puedes suponer, la ejecución continúa con las sentencias escritas tras la estructura de control (tras el **end**).

Por ejemplo, la función **valordigitohex** devuelve como resultado el valor de un dígito hexadecimal. En base 16, llamada hexadecimal, se utilizan los dígitos del 0 al 9 normalmente y se utilizan las letras de la *A* a la *F* para representar los dígitos con valores del 10 al 15. Esta función soluciona el problema por casos, seleccionando el caso en función del valor de **digito**. La hemos programado algo más complicada de lo que podría ser, para que veas varias formas de escribir valores en las ramas de un **case**.

```
 1 {
 2     valor de digito hexadecimal
 3 }
 4 program valorhex;
 5
 6 function valordigitohex(digito: char): integer;
 7 begin
 8     { mas ramas de las necesarias para que tengas ejemplos }
 9     case digito of
10     '0':
11         result := 0;
12     '1'..'9':
13         result := ord(digito)-ord('0');
14     'A','B','C'..'F':
15         begin
16             result := 10+ord(digito)-ord('A');
17         end;
18     'a'..'f':
19         result := 10+ord(digito)-ord('a');
20     otherwise
21         result := 0;
22     end;
23 end;
24
```

```
25 begin
26     writeln(valordigitohex('E'));
27 end.
28
```

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

```
'0':
   result := 0;
```

hace que la función devuelva 0 cuando **digito** es . La segunda rama es similar, pero muestra una forma de ejecutar la misma rama para varios valores distintos que son consecutivos.

```
'1'..'9':
   result := ord(digito)-ord('0');
```

En este caso, cuando *digito* esté entre "**1**'' y "**9**'' se ejecutará esta rama.

A la expresión que utiliza varios valores separados por "**..**'' la denominamos **rango** y la podemos entender como el conjunto de valores que están entre los dos indicados, incluyendo ambos.

La tercera rama muestra como podemos separar por comas valores no consecutivos, para ejecutar una sentencia para diferentes valores. Además, podemos ver que es posible utilizar un rango junto con valores individuales.

La rama **otherwise** ejecuta cuando ninguna otra lo hace. Incluso si pensamos que no podríamos adoptar ningún otro valor, pero los casos no cubren todos los valores posibles, deberíamos incluir esta rama y hacer que el programa haga lo que sea preferible si tal cosa ocurre. Quizá, imprimir un mensaje y terminar de ejecutar tras informar del error.

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

La función se habría podido escribir de un modo más compacto como sigue.

```
1 {
2     valor de digito hexadecimal
3 }
4 program valorhex;
```

```
 5
 6 function valordigitohex(digito: char): integer;
 7 begin
 8     case digito of
 9     '0'..'9':
10         result := ord(digito)-ord('0');
11     'a'..'f':
12         result := 10+ord(digito)-ord('a');
13     'A'..'F':
14         result := 10+ord(digito)-ord('A');
15     otherwise
16         writeln('ERROR: valordigitohex: digito: ', digito);
17         halt(1);
18         result := 0;
19     end;
20 end;
21
22 begin
23     writeln(valordigitohex('Z'));
24 end.
25
```

Como puedes ver, esta vez hacemos que si se utiliza la función con un valor incorrecto el programa se detenga tras avisarnos. Recuerda que las funciones **no deben escribir** ni leer nada, tan sólo utilizar sus parámetros. No obstante, para depurar errorres lo mejor es informar del problema que tenemos y detener el programa cuanto antes, para que sea mas simple encontrar el error.

Otro detalle es que además le damos un valor a la función, por si alguna vez cambiamos el código, para que no se olvide que hemos de indicar qué valor adopta la función en esa rama.

4.4. Punto más distante a un origen

Supongamos que tenemos valores discretos dispuestos en una línea y tomamos uno de ellos como el origen. Nos puede interesar ver cuál de dos valores es el más distante al origen. Por ejemplo, en la figura 2, ¿Será el punto*a* o el *b* el más lejano al origen *o*?

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

```
function masdistante(a, b: integer): integer;
begin
```

Fig. 2. Dos puntos *a* y *b* en una recta discreta con origen en *o*.

```
    if distancia(Origen, a) > distancia(Origen, b) then begin
        result := a;
    end
    else begin
        result := b;
    end;
end;
```

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

```
function distancia(p1, p2: integer): integer;
begin
    if p1 > p2 then begin
        result := p1-p2;
    end
    else begin
        result := p2-p1;
    end;
end;
```

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

```
function distancia(p1, p2: integer): integer;
begin
    result := (p1-p2) * signo(p1-p2);
end;
```

dado que al multiplicar la diferencia por su signo obtenemos la diferencia en valor absoluto. Y una tercera forma sería utilizar la función **abs** que devuelve el valor absoluto.

El programa completo quedaría como sigue.

```
1
2 {
```

```
 3     punto mas distante a un origen
 4
 5 }
 6 program distancias;
 7
 8 const
 9     Origen: integer = 0;
10
11 function distancia(p1, p2: integer): integer;
12 begin
13     if p1 > p2 then begin
14         result := p1-p2;
15     end
16     else begin
17         result := p2-p1;
18     end;
19 end;
20
21 function masdistante(a, b: integer): integer;
22 begin
23     if distancia(Origen, a) > distancia(Origen, b) then begin
24         result := a;
25     end
26     else begin
27         result := b;
28     end;
29 end;
30
31 const
32     P1: integer = 3;
33     P2: integer = -2;
34
35 begin
36     writeln(masdistante(P1, P2));
37 end.
38
```

4.5. Mejoras

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

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

```
function espositivo(n: integer): boolean;
begin
```

```
    if n > 0 then begin
        result := True;
    end
    else begin
        result := False;
    end;
end;
```

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

```
function espositivo(n: integer): boolean;
begin
    result := n > 0;
end;
```

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

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

```
 1 program writes;
 2
 3 const
 4     A: integer = 10;
 5     B: integer = -3;
 6 begin
 7     if A > B then begin
 8         write('el numero ');
 9         write(A);
10         write(' es mayor que ');
11         write(B);
12         writeln();
13     end
14     else begin
15         write('el numero ');
16         write(B);
17         write(' es mayor que ');
18         write(B);
19         writeln();
20     end;
```

```
21
22 end.
```

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

```
 1 program writes;
 2
 3 const
 4     A: integer = 10;
 5     B: integer = -3;
 6 begin
 7     write('el numero ');
 8     if A > B then begin
 9         write(A);
10         write(' es mayor que ');
11         write(B);
12         writeln();
13     end
14     else begin
15         write(B);
16         write(' es mayor que ');
17         write(B);
18         writeln();
19     end;
20
21 end.
```

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

```
 1 program writes;
 2
 3 const
 4     A: integer = 10;
 5     B: integer = -3;
 6 begin
 7     write('el numero ');
 8     if A > B then begin
 9         write(A);
10         write(' es mayor que ');
11         write(B);
12     end
```

```
13      else begin
14          write(B);
15          write(' es mayor que ');
16          write(B);
17      end;
18      writeln();
19
20  end.
```

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

4.6. Primeros tres dígitos hexadecimales

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

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

```
writeln(digitohex(valordigito(Num, 3, 16)));
writeln(digitohex(valordigito(Num, 2, 16)));
writeln(digitohex(valordigito(Num, 1, 16)));
```

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

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

Primero definir el problema: suministramos un entero y queremos un carácter

que corresponda con el dígito. Luego la cabecera de la función ha de ser:

```
function digitohex(valor: integer): char;
```

Esta función es un caso típico de un problema de casos. Si el valor está entre 0 y 9 entonces podemos sumar el valor a la posición que ocupa el "**0**'' en el código de caracteres y de esta forma obtener el código para el carácter correspondiente. Convirtiendo ese código a un carácter tenemos la función resuelta. En otro caso, el valor ha de estar entre 10 y 15 y lo que tenemos que hacer es restar 10 al valor entero, y hacer lo mismo que antes, pero desde el carácter **A**. Por ejemplo, para el valor 12, devolveríamos el carácter correspondiente al código de la letra **A** mas 2, esto es, la **C**:

```
function digitohex(valor: integer): char;
begin
    if valor >= 10 then begin
        result := char(ord('A')+valor-10);
    end
    else begin
        result := char(ord('0')+valor);
    end;
end;
```

Este es el programa completo.

```
{
    primeros tres digitos hexa
}
program hexdigitos;

uses math;    {por ** }

function valordigito(n, pos, base: integer): integer;
begin
    result := (n div base**(pos-1)) mod base;
end;

function digitohex(valor: integer): char;
begin
    if valor >= 10 then begin
        result := char(ord('A')+valor-10);
    end
    else begin
        result := char(ord('0')+valor);
    end;
end;

const
```

```
24      Num: integer = 256+15;
25      Base: integer = 16;
26
27  begin
28      writeln(digitohex(valordigito(Num, 3, Base)));
29      writeln(digitohex(valordigito(Num, 2, Base)));
30      writeln(digitohex(valordigito(Num, 1, Base)));
31  end.
```

4.7. ¿Es válida una fecha?

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

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

```
 1  {
 2      es valida una fecha?
 3  }
 4  program fechavalida;
 5
 6  const
 7      MinAnyo: integer = 1;
 8      MaxAnyo: integer = 3000;
 9
10      Ene: integer = 1;
11      Dic: integer = 12;
12
13  function anyook(anyo: integer): boolean;
14  begin
15      result := (anyo >= MinAnyo) and (anyo <= MaxAnyo);
16  end;
17
18  function mesok(mes: integer): boolean;
19  begin
20      result := (mes >= Ene) and (mes <= Dic);
21  end;
22
23
24  function diaok(dia: integer): boolean;
25  begin
```

```
26      result := True;
27  end;
28
29  function fechaok(anyo, mes, dia: integer): boolean;
30  begin
31      result := anyook(anyo) and mesok(mes) and diaok(dia);
32  end;
33
34  const
35      Anyo: integer = 2001;
36      Mes: integer = 11;
37      Dia: integer = 16;
38
39  begin
40      writeln(fechaok(Anyo, Mes, Dia));
41  end.
```

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

`function diaok(anyo, mes, dia: integer): boolean;`

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

La función **`diaok`** parece por lo demás un caso típico de problema por casos. Hay meses que tienen 30 y hay meses que 31. El caso de febrero va a ser un poco más complicado. Parece una función a la medida de un **`case`**, dado que según el valor del mes tenemos un caso u otro de entre 12 casos distintos. Además, necesitamos una función auxiliar que nos diga si el año en cuestión es bisiesto o no. Pero esa ya la teníamos, o casi.

```
function esbisiesto(anyo: integer): boolean;
begin
    result := (Anyo mod 4 = 0) and (Anyo mod 100 <> 0) or
        (Anyo mod 400 = 0);
end;

function diaok(anyo, mes, dia: integer): boolean;
begin
    case mes of
    1, 3, 5, 7, 8, 10, 12:
        result := (dia >= 1) and (dia <= 31);
    2:
```

```
        if esbisiesto(anyo) then begin
            result := (dia >= 1) and (dia <= 29);
        end
        else begin
            result := (dia >= 1) and (dia <= 28);
        end;
    otherwise
        result := (dia >= 1) and (dia <= 30);
    end;
end;
```

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

El programa completo queda como sigue.

```
 1 {
 2     es valida una fecha?
 3 }
 4 program fechavalida;
 5
 6 const
 7     MinAnyo: integer = 1;
 8     MaxAnyo: integer = 3000;
 9
10     Ene: integer = 1;
11     Dic: integer = 12;
12
13 function anyook(anyo: integer): boolean;
14 begin
15     result := (anyo >= MinAnyo) and (anyo <= MaxAnyo);
16 end;
17
18 function mesok(mes: integer): boolean;
19 begin
20     result := (mes >= Ene) and (mes <= Dic);
21 end;
22
23 function esbisiesto(anyo: integer): boolean;
24 begin
25     result := (Anyo mod 4 = 0) and (Anyo mod 100 <> 0) or
26         (Anyo mod 400 = 0);
27 end;
28
```

```
29 function diasenmes(anyo, mes: integer): integer;
30 begin
31     case mes of
32     1, 3, 5, 7, 8, 10, 12:
33         result := 31;
34     2:
35         if esbisiesto(anyo) then begin
36             result := 29
37         end
38         else begin
39             result := 28;
40         end;
41     otherwise
42         result := 30;
43     end;
44 end;
45
46 function diaok(anyo, mes, dia: integer): boolean;
47 begin
48     result := (dia >= 1) and (dia <= diasenmes(anyo, mes));
49 end;
50
51 function fechaok(anyo, mes, dia: integer): boolean;
52 begin
53     result := anyook(anyo) and
54             mesok(mes) and diaok(anyo, mes, dia);
55 end;
56
57 const
58     Anyo: integer = 2001;
59     Mes: integer = 11;
60     Dia: integer = 16;
61
62 begin
63     writeln(fechaok(Anyo, Mes, Dia));
64 end.
```

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

4.8. Problemas

Escribe un programa en Pascal que calcule cada uno de los siguientes problemas (Para aquellos enunciados que corresponden a problemas ya hechos te sugerimos

que los hagas de nuevo pero esta vez sin mirar la solución).

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

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

3. Valor numérico de un dígito hexadecimal.
4. Distancia entre dos puntos en una recta discreta.
5. Ver si número entero es positivo.
6. Máximo de dos números enteros.
7. Máximo de tres números enteros
8. Convertir un carácter a mayúscula, dejándolo como está si no es una letra minúscula.
9. Devolver el número de círculos que tiene un dígito (gráficamente al escribirlo: por ejemplo, 0 tiene un círculo, 2 no tiene ninguno y 8 tiene dos).
10. Indicar el cuadrante en que se encuentra un número complejo dada su parte real e imaginaria.
11. Devolver un valor de verdad que indique si un número de carta (de 1 a 10) corresponde a una figura.
12. Devolver el valor de una carta en el juego de las 7 y media.
13. Devolver el número de días del mes teniendo en cuenta si el año es bisiesto.
14. Indicar si un número de mes es válido o no.
15. Indicar si una fecha es válida o no.
16. Ver si un número puede ser la longitud de la hipotenusa de un triángulo rectángulo de lados dados.
17. Decidir si la intersección de dos rectángulos es vacía o no.
18. Decidir si se aprueba una asignatura dadas las notas de teoría y práctica teniendo en cuenta que hay que aprobarlas por separado (con notas de 0 a 5 cada una) pero pensando que se considera que un 4.5 es un 5 en realidad.
19. Devolver el primer número impar mayor o igual a uno dado.
20. Devolver el número de días desde comienzo de año dada la fecha actual.
21. La siguiente letra a una dada pero suponiendo que las letras son circulares (esto es, detrás de la vendría de nuevo la).
22. Resolver una ecuación de segundo grado dados los coeficientes, sean las raíces de la ecuación reales o imaginarias.
23. Determinar si tres puntos del plano forman un triángulo. Recuerda que la condición para que tres puntos puedan formar un triángulo es que se cumpla la condición de

triangularidad. Esto es, que cada una de las distancias entre dos de ellos sea estrictamente menor que las distancias correspondientes a los otros dos posibles lados del triángulo.

Índice