Introducción a la Programación usando Pascal como primer lenguaje

Capítulo 3
Resolución de problemas

Francisco J. Ballesteros

©2018 Francisco J. Ballesteros

Resolución de problemas

3.1. Problemas y funciones

El propósito de un programa es resolver un problema. Y ya hemos resuelto algunos. Por ejemplo, en el capítulo anterior utilizamos una función para calcular la longitud de una circunferencia. Ahora vamos a prestar más atención a lo que hicimos para ver cómo resolver problemas nuevos y problemas más difíciles.

Lo primero que necesitábamos al realizar un programa era definir el problema. Pensando en esto, prestemos atención ahora a esta línea de código que ya vimos antes:

```
function longitudcircunferencia(r: real): real;
```

Esta línea forma parte de la definición de la función **`longitudcircunferencia`** en Pascal y se la denomina la **cabecera de función**. En este caso, es la cabecera para la función **`longitudcircunferencia`**. El propósito de esta línea es indicar:

1. Cómo se llama la función.
2. Qué necesita la función para hacer sus cálculos.
3. Qué devuelve la función como resultado (qué tipo de datos devuelve la función).

Si lo piensas, ¡La definición de un problema es justo esto!

1. Cuál es el problema que resolvemos.
2. Qué necesitamos para resolverlo.
3. Qué tendremos una vez esté resuelto.

Luego

la definición de un problema es la cabecera de una función

En general, decimos que la definición de un problema es la cabecera de un subprograma (dado que hay también otro tipo de subprogramas, llamados procedimientos, como veremos más adelante). Esto es muy importante debido a que lo vamos a utilizar a todas horas mientras programamos.

Por ejemplo, si queremos calcular el área de un círculo, nuestro problema es: dado un radio r, calcular el valor de su área (r^2). Dicho de otro modo, el problema consiste en tomar un número real,r, y calcular otro n úmero real.

El nombre del problema en Pascal podría ser **`areacirculo`**. Si queremos programar este problema, ya sabemos que al menos tenemos que definir una función con este nombre:

```
function areacirculo
```

Puede verse que el identificador que da nombre a la función se escribe tras la palabra reservada **`function`**. Dicho identificador debe ser un nombre descriptivo del resultado

de la función (¡Un nombre descriptivo del problema!).

Ahora necesitamos ver qué necesitamos para hacer el trabajo. En este caso el radio *r* , que es un número real. En Pascal podríamos utilizar el identificador **`r`** (dado que en este caso basta para saber de qué hablamos). Además, sabemos ya que este valor tiene que ser de tipo **`real`**.

La forma de suministrarle valores a la función para que pueda hacer su trabajo es definir sus **parámetros**. En este caso, un único parámetro **`r`** de tipo **`real`**. Los parámetros hay que declararlos o definirlos entre paréntesis, tras el nombre de la función (Piensa que en matemáticas usarías$f(x)$ par a una función de un parámetro). Resumiendo, por el momento sabemos que hay que escribir

```
function areacirculo(r: real)
```

Por último necesitamos definir de qué tipo va ser el valor resultante cada vez que utilicemos la función. En este caso la función tiene como valor un número real. Decimos que devuelve un número real. En una cabecera de Pascal, el tipo de dato devuelto se especifica poniendo dos puntos y el tipo, detrás de los parámetros. Y para finalizar la cabecera, es preciso utilizar un "**`;`**'' como en el resto de declaraciones.

```
function areacirculo(r: real): real;
```

¡Esta línea es la definición en Pascal de nuestro problema!

Hemos dado el primer paso de los que teníamos que dar para resolverlo. En cuanto lo hayamos hecho podremos escribir expresiones como **`areacirculo(3.2)`** para calcular el área del circulo con radio 3. 2. Al hacerlo, suministramos 3.2 como valor para **`r`** en la función (véase la figura 1); El resultado de la función es otro valor real (en este caso ar^2).

Deberás recordar que que se llama **argumento** al valor concreto que se suministra a la función (por ejemplo, 3.2) cuando se produce una llamada (cuando se evalúa la función) y **parámetro** al identificador que nos inventamos para nombrar el argumento que se suministrará cuando se produzca una llamada (por ejemplo En la figura 1 los argumentos están representados por valores sobre las flechas que se dirigen a los parámetros (los cuadrados pequeños a la entrada de cada función).

En muchas ocasiones un problema requerirá de varios elementos para su resolución. Por ejemplo, obtener la media de dos números requiere dos números. Si queremos la media de 1.3 y 4.2 podríamos escribir **`media(1.3,4.2)`**. En tal caso se procede del mismo modo para definir el problema (la cabecera de la función), pero separamos cada parámetro con un "**`;`**'' al escribir en Pascal. Por ejemplo:

```
function media(num1: real; num2: real): real;
```

Si varios parámetros tienen el mismo tipo, podemos abreviar un poco separando los nombres con "**`,`**'' y usando el mismo tipo:

```
function media(num1, num2: real): real;
```

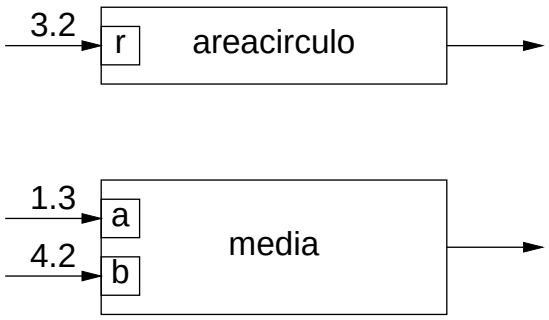

Fig. 1. Dos funciones en Pascal. Tienen argumentos y devuelven resultados.

Volviendo a nuestro problema de ejemplo, resta implementar la solución. La forma de hacer esto en Pascal es escribir el **cuerpo** de la función. Para hacerlo supondremos que tenemos ya definida la constante **`Pi`**. Si no es así, como ahora nos vendría bien tenerla, nos la inventamos justo ahora y no hay mayor problema. La función con su cabecera y cuerpo queda como sigue:

```
function areacirculo(r: real): real;
begin
    result := Pi*sqr(r);
end;
```

La sentencia que tenemos en el cuerpo hace que la función tome un valor como resultado. El Pascal se permite también utilizar el nombre de la función en lugar de la palabra reservada **result**. Pero en el dialecto de Pascal que utilizamos es mas claro para nosotros utilizar **result** para evitar errores.

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

```
 1
 2 {
 3     Area de circulo
 4 }
 5 program areacirc;
 6
 7 const
 8     Pi: real = 3.14;
 9
10 function areacirculo(r: real): real;
11 begin
12         result := Pi*sqr(r);
13 end;
```

```
14
15 {constantes de prueba aqui}
16 const
17     Radio1: real = 3.2;
18     Radio2: real = 4.0;
19 begin
20     writeln(areacirculo(Radio1));
21     writeln(areacirculo(Radio2));
22 end.
```

Dado que la función es obvio lo que hace, lo que necesita y lo que devuelve a la luz de los nombres que hemos utilizado, nos parece innecesario escribir un comentario aclarando lo que hace la función.

Al compilar el programa, se definieron las constantes. Al ejecutar este programa, se comenzarán a ejecutar las sentencias del cuerpo del programa principal. Cuando llegue el momento de evaluar **`areacirculo(Radio1)`**, se "llamará'' a la función **`areacirculo`** utilizando **`Radio1`** (esto es, 3.2) como argumento (como valor) para el parámetro **`r`**. La función tiene como cuerpo una única sentencia que hace que el valor resultante de la función sea r^2. Después ejecutará la segunda sentencia del programa principal, que llama a la función usando **`Radio2`** como argumento (cuyo valor es 4.0). Cuando compilamos y ejecutamos el programa, podemos ver:

```
unix% fpc areacirc.p
unix% areacirc
3.2153600000000004E+001
5.0240000000000002E+001
```

Podemos utilizar llamadas a función en cualquier lugar donde podemos utilizar un valor del tipo que devuelve la función considerada (salvo para declarar constantes). Por ejemplo, **`areacirculo(x)`** puede utilizarse en cualquier sitio en que podamos emplear un valor de tipo **`real`** que tengamos en algo llamado **`x`**.

Sólo podemos utilizar la función a partir del punto en el programa en que la hemos definido. Nunca antes. Lo mismo sucede con cualquier otra declaración.

Veamos un ejemplo. Supongamos que en alguna sentencia del programa principal utilizamos la expresión

```
(3.0 + areacirculo(2.0 * 5.0 - 1.0)) / 2.0
```

Cuando Pascal encuentre dicha expresión la evaluará, procediendo como hemos visto para otras expresiones:

```
(3.0 + areacirculo(2.0 * 5.0 - 1.0)) / 2.0
(3.0 + areacirculo(10.0 - 1.0)) / 2.0
(3.0 + areacirculo(9.0)) / 2.0
```

En este punto Pascal deja lo que está haciendo y llama a **`areacirculo`** utilizando **`9.0`**

como argumento (como valor) para el parámetro *r*. Eso hace que Pascal ejecute el cuerpo de la función y calcule

```
Pi*sqr(r)
```

lo que tiene como valor **`254.469025`**. La sentencia que comienza por "**`result :=`**'' hace que se la función tome este valor. Al llegar al "**`end`**'' de la función, Pascal retorna de la función. Ahora Pascal continúa con lo que estaba haciendo en el punto en que se llamó a la función, evaluando nuestra expresión:

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

Resumiendo todo esto, podemos decir que una función en Pascal es similar a una función matemática. A la función matemática se le suministran valores de un conjunto origen y la función devuelve como resultado valores en otro conjunto imagen. En el caso de Pascal, una función recibe uno o más valores y devuelve un único valor de resultado. Por ejemplo, si **`areacirculo`** es una función que calcula el área de un circulo dado un radio, entonces **`areacirculo(3.0)`** es un valor que corresponde al área del círculo de radio 3. Cuando Pascal encuentra **`areacirculo(3.0)`** en una expresión, procede a evaluar la función llamada **`areacirculo`** y eso deja como resultado un valor devuelto por la función.

Desde este momento sabemos que

un subproblema lo resuelve un subprograma

y definiremos un subprograma para cada subproblema que queramos resolver. Hacerlo así facilita la programación, como ya veremos. Por supuesto necesitaremos un programa principal que tome la iniciativa y haga algo con el subprograma (llamarlo), pero nos vamos a centrar en los subprogramas la mayor parte del tiempo.

En muchos problemas vamos a tener parámetros que están especificados por el enunciado (como "100'' en "calcular los 100 primeros números primos''). En tal caso lo mejor es definir constantes para los parámetros del programa. La idea es que

para cambiar un dato debe bastar cambiar una constante

y volver a compilar y ejecutar el enunciado. Esto no sólo es útil para modificar el programa, también elimina errores.

3.2. Declaraciones y definiciones

Hemos estado utilizando declaraciones todo el tiempo sin saber muy bien lo que es esto y sin prestar mayor atención a este hecho. Es hora de verlo algo más despacio. Normalmente se suele distinguir entre **declarar** algo en un programa y **definir** ese algo. Por ejemplo, la cabecera de una función *declara* que la función existe, tiene

unos parámetros y un resultado. El cuerpo de una función *define* cómo se calcula la función.

En el caso de las constantes que hemos estado utilizando durante el curso, las líneas del programa que las *declaran* están también definiéndolas (dado que definen qué valor toman).

Un objeto (una contante, un subprograma, un tipo, etc.) sólo puede utilizarse desde el punto en que se ha declarado hasta el final del programa (o subprograma) en que se ha declarado. Fuera de este **ámbito** o zona del programa el objeto no es **visible** y es como si no existiera. Volveremos sobre esto más adelante.

Habitualmente resulta útil declarar constantes cuando sea preciso emplear constantes bien conocidas (o literales que de otro modo parecerían números mágicos o sin explicación alguna). Para declarar una constante se procede como hemos visto durante los capítulos anteriores. La declaración de constantes se realiza en una sección del programa especial, que comienza con la palabra reservada *const* seguida de todas las definiciones de constantes que necesitemos. Por ejemplo:

```
const
    Pi: real = 3.1415926;
    Maximo: integer = 3;
    Inicial: char = 'B';
```

Para definir una constante en la sección, primero se escribe el identificador de la constante, por ejemplo **`Pi`**, seguido de "**`:`**'' y el nombre del tipo para la constante, seguido de "**`=`**'' y seguido del valor de la constante. Como de costumbre, también hay que escribir un "**`;`**'' para terminar la declaración. Si omitimos el tipo, la constante tendrá el tipo correspondiente al valor que se le ha asignado en su definición, pero este no siempre será el tipo que esperamos (hay diversos tipos de enteros, por ejemplo).

Una vez hemos declarado la constante **`Pi`** podemos utilizar su identificador en expresiones en cualquier lugar del programa desde el punto en que se ha declarado la constante hasta el final del programa.

3.3. Problemas con solución directa

Hay muchos problemas que ya podemos resolver mediante un programa. Concretamente, todos los llamados *problemas con solución directa*. Como su nombre indica son problemas en que no hay que decidir nada, tan sólo efectuar un cálculo según indique algún algoritmo o teorema ya conocido.

Por ejemplo, la siguiente función puede utilizarse como (sub)programa para resolver el problema de calcular el factorial de 5.

```
8 function fact5(): real;
9 begin
```

```
10     result := 5*4*3*4*1;
11 end;
```

Ha bastado seguir estos pasos:

1. Buscar la definición de factorial de un número.
2. Definir el problema en Pascal (escribiendo la cabecera de la función).
3. Escribir la expresión en Pascal que realiza el cálculo, usando la sentencia del cuerpo de la función.
4. Compilarlo y probarlo.
5. Depurar los errores cometidos.

Procederemos siempre del mismo modo.

Para problemas más complicados hay que emplear el optimismo. La clave de todo radica en suponer que tenemos disponible todo lo que nos pueda hacer falta para resolver el problema que nos ocupa (salvo la solución del problema que nos ocupa, claro está). Una vez resuelto el problema, caso de que lo que nos ha hecho falta para resolverlo no exista en realidad, tenemos que proceder a programarlo. Para ello volvemos a aplicar la misma idea.

Esto se entenderá fácilmente si intentamos resolver un problema algo más complicado. Supongamos que queremos calcular el valor del volumen de un sólido que es un cilindro al que se ha perforado un hueco cilíndrico, tal y como muestra la figura 2.

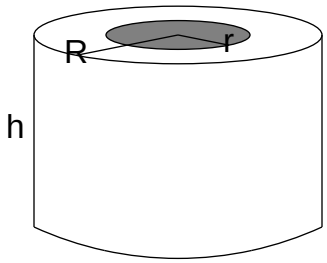

Fig. 2. Un sólido cilíndrico perforado por un cilindro.

Lo primero es definir el problema. ¿Qué necesitamos saber para que esté definido nuestro sólido? Necesitamos la altura del cilindro, *h* en la figura. También el radio del cilindro que hemos taladrado, *r* en la figura, y el radio del cilindro, *R* en la figura. ¿Qué debemos producir como resultado? El valor del volumen. Luego...

```
{
  volumen de un cilindro de altura alt y radio rmax
  taladrado por otro cilindro de radio rmin
}
function volcilhueco(rmin, rmax, alt: real): real;
```

es la definición de nuestro problema. Donde hemos utilizado **`alt`** para *h*, **`rmin`** para *r*, y **`rmax`** para *R*.

¡Y ahora somos optimistas!

Si suponemos que ya tenemos calculado el área de la base, basta multiplicarla por la altura para tener nuestro resultado. En tal caso, lo suponemos y hacemos nuestro programa.

```
function volcilhueco(rmin, rmax, alt: real): real;
begin
    result := ¿Area de la base? * alt;
end;
```

El área de la base es en realidad el área de una corona circular, de radios **`rmin`** y **`rmax`** (o *r* y *R*). Luego el problema de calcular el área de la base es el problema de calcular el área de una corona circular (como la que muestra la figura 3).

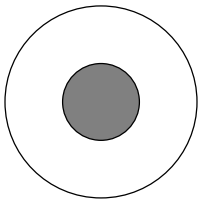

Fig. 3. Una corona circular (un círculo con un agujero circular en su centro)

Este problema lo definimos en Pascal como la cabecera de función:

```
function areacorona(rmin: real; rmax: real): real;
```

Por lo que en realidad nuestro programa debe ser ahora:

```
function volcilhueco(rmin, rmax, alt: real): real;
begin
    result := areacorona(rmin, rmax) * alt;
end;
```

Y ya tenemos programado nuestro problema. Bueno, en realidad ahora tendremos que programar **areacorona**.

Siendo optimistas de nuevo, si suponemos que tenemos programado el cálculo del área de un círculo, podemos calcular el área de una corona circular como la diferencia entre el área del circulo externo y el área del circulo taladrado en su centro. Luego...

```
function areacorona(rmin: real; rmax: real): real;
begin
    result := areacirculo(rmax) - areacirculo(rmin);
end;
```

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

```
 1
 2 {
 3     volumen cilindro hueco
 4 }
 5 program fact;
 6
 7 const
 8     Pi: real = 3.1415926;
 9
10 function areacirculo(r: real): real;
11 begin
12     result := Pi*sqr(r);
13 end;
14
15 function areacorona(rmin: real; rmax: real): real;
16 begin
17     result := areacirculo(rmax) - areacirculo(rmin);
18 end;
19
20 {
21   volumen de un cilindro de altura alt y radio rmax
22   taladrado por otro cilindro de radio rmin
23 }
24 function volcilhueco(rmin, rmax, alt: real): real;
25 begin
26     result := areacorona(rmin, rmax) * alt;
27 end;
```

```
28
29 const
30     Rmin: real = 3.2;
31     Rmax: real = 4.3;
32     Alt: real = 2.8;
33
34 begin
35     writeln('v = ', volcilhueco(Rmin, Rmax, Alt) :0:3);
36 end.
```

Un último apunte respecto a los parámetros. Aunque los mismos nombres **`rmin`** y **`rmax`** se han utilizado para los parámetros en varias funciones distintas esto no tiene por qué ser así. Dicho de otra forma: el parámetro **`rmin`** de **`areacorona`** no tiene nada que ver con el parámetro **`rmin`** de **`volcilhueco`**¡Aunque tengan los mismos nombres!

Esto no debería ser un problema. Piensa que en la vida real personas distintas identifican objetos distintos cuando hablan de “el coche’’, aunque todos ellos utilicen la misma palabra (“coche’’). En nuestro caso sucede lo mismo: cada función puede utilizar los nombres que quiera para sus parámetros.

3.4. Subproblemas

Lo que acabamos de hacer para realizar este programa se conoce como **refinamiento progresivo**. Consiste en solucionar el problema poco a poco, suponiendo cada vez que tenemos disponible todo lo que podamos imaginar (salvo lo que estamos programando, claro). Una vez programado nuestro problema nos centramos en programar los subproblemas que hemos supuesto que teníamos resueltos. Y así sucesivamente. Habrás visto que

un subproblema se resuelve con un subprograma

Sí, ya lo habíamos enfatizado antes. Pero esto es muy importante. En este caso hemos resuelto el problema comenzando por el problema en si mismo y descendiendo a subproblemas cada vez más pequeños. A esto se lo conoce como desarrollo **top-down** (de arriba hacia abajo). Es una forma habitual de programar.

En la práctica, ésta se combina con pensar justo de la forma contraria. Esto es, si vamos a calcular volúmenes y áreas y hay figuras y sólidos circulares, seguro que podemos imaginarnos ciertos subproblemas elementales que vamos a tener que resolver. En nuestro ejemplo, a lo mejor podríamos haber empezado por resolver el problema **`areacirculo`**. Podríamos haber seguido después construyendo programas más complejos, que resuelven problemas más complejos, a base de utilizar los que ya teníamos. A esto se lo conoce como desarrollo **bottom-up** (de abajo hacia arriba).

Para programar hay que combinar las dos técnicas. Hay que organizar el

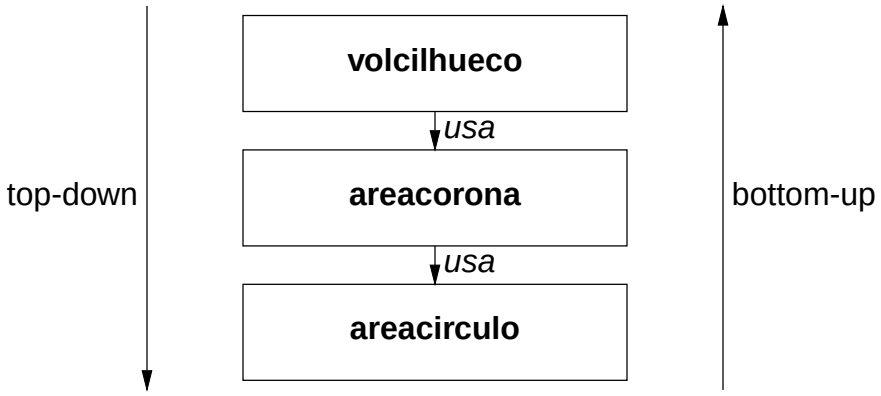

Fig. 4. Podemos programar de arriba-a-abajo o de abajo-a-arriba. En realidad, hay que usar ambas.

programa pensando *top-down* aunque seguramente lo programemos *bottom-up*. Dicho de otro modo, abordaremos los problemas dividiéndolos en subproblemas más sencillos, pero normalmente sin programar en absoluto, hasta que tengamos un subproblema tan simple que sea fácil de programar. En ese momento lo programamos y, usándolo, continuamos programando subproblemas cada vez más complejos hasta resolver el problema original. Esto lo iremos viendo a lo largo de todo el curso.

3.5. Algunos ejemplos

Vamos a ver a continuación algunos ejemplos. En algunos casos sólo incluiremos una función que calcule el problema, y no el programa por completo. Ya sabemos cómo utilizar funciones en programas Pascal y no debería haber problema en conseguir ejecutar éstos programas.

3.5.1. Escribir dígitos con espacios

Queremos un programa que escriba un número de tres dígitos con cada uno en una línea.

Lo primero que tenemos que hacer es ver si existe algún subprograma que, de estar disponible, nos deje hacer el trabajo de forma trivial. En este caso, de tener una función que nos suministre el tercer dígito, otra que nos suministre el segundo y otra que nos suministre el primero, bastaría usar estas tres y luego utilizar **`writeln`** para escribir por pantalla cada uno de los dígitos.

Pues bien, supondremos que tenemos estas funciones. De hecho, vamos a suponer que tenemos *una* función que es capaz de darnos el dígito n-ésimo de un número. A esta función la vamos a llamar **`valordigito`**. De este modo, en lugar de

tener que programar tres funciones distintas vamos a tener que programar una única función.

Considerando esta función como un subproblema, vemos que tenemos que suministrarle tanto el número como la posición del dígito que nos interesa. Una vez la tengamos podremos escribir nuestro programa.

Vamos a hacer justo eso para empezar a resolver el problema: vamos a escribir nuestro programa utilizando dicha función, aunque por el momento la vamos a programar para que siempre devuelva cero como resultado (lo que es fácil).

```
 1 {
 2     escribir digitos separados
 3 }
 4 program digitos;
 5 
 6 function valordigito(n, pos: integer): integer;
 7 begin
 8     result := 0;
 9 end;
10 
11 const
12     Num: integer = 134;
13 begin
14     writeln(valordigito(Num, 3));
15     writeln(valordigito(Num, 2));
16     writeln(valordigito(Num, 1));
17 end.
```

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

```
unix% fpc digitos.p
unix% digitos
0
0
0
```

Todo bien.

Pasemos a implementar la parte que nos falta. Ahora queremos el n-ésimo dígito de la parte entera de un número. Por ejemplo, dado 134, si deseamos el segundo dígito entonces queremos obtener 3 como resultado.

Si el problema parece complicado (aunque no lo es), lo primero que hay que hacer es **simplificarlo**. Por ejemplo, suponemos que siempre vamos a querer el segundo dígito.

Si sigue siendo complicado, lo volvemos a simplificar. Por ejemplo, supondremos que sólo queremos el primer dígito. Este parece fácil. Sabemos que cada dígito corresponde al factor de una potencia de 10 en el valor total del número. Esto es,

$$143 = 1 \times 10^2 + 4 \times 10^1 + 3 \times 10^0$$

Por lo tanto, el módulo entre 10 es en realidad el valor del primer dígito. Una vez que hemos visto esto, podemos complicar el problema un poco más para resolver algo más cercano al problema original.

Consideremos el problema que habíamos considerado antes consistente en obtener el segundo dígito. Si dividimos entre 10 obtendremos el efecto de desplazar los dígitos a la derecha una unidad. Tomando a continuación el módulo entre 10 obtendremos el primer dígito, que era antes el segundo. Si queremos el tercer dígito habríamos tenido que dividir por 100 y tomar luego el módulo entre 10.

Dado que esto parece fácil ahora, podemos complicarlo un poco más y obtener una solución para nuestro problema original. Si dividimos entre 10^{n-1} y tomamos el módulo entre 10 tenemos el valor del dígito que nos interesa. En Pascal podemos programar esto como una expresión de forma directa. Así pues, basta cambiar la función **`valordigito`** para dejar el programa como sigue.

```
 1 {
 2     escribir digitos separados
 3 }
 4 program digitos;
 5 
 6 uses math;
 7 
 8 function valordigito(n, pos: integer): integer;
 9 begin
10     result := (n div 10**(pos-1)) mod 10;
11 end;
12 
13 const
14     Num: integer = 134;
15 begin
16     writeln(valordigito(Num, 3));
17     writeln(valordigito(Num, 2));
18     writeln(valordigito(Num, 1));
19 end.
```

Y de este modo tendremos el programa completo. Si ahora lo compilamos y lo

ejecutamos veremos que, en lugar de escribir ceros, el programa escribe dígitos.

```
unix% fpc digitos.p
unix% digitos
1
3
4
```

¿Tenemos el programa hecho? Bueno, en principio sí. Pero merece la pena pensar un poco más y ver si podemos hacer algo mejor.

Lo único que no gusta mucho en este código es que aparezcan literales tales como "**10**" de vez en cuando. Estos **números mágicos** que aparecen en los programas los hacen misteriosos. Tal vez no mucho en este caso, pero en general es así.

El "**10**" que aparece varias veces en el código es la base del sistema de numeración. Una opción sería cambiar la función por esta otra, declarando antes una constante **Base** para la base que utilizamos:

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

No obstante, puede que un mismo programa necesite utilizar una función como esta pero empleando bases de numeración distintas cada vez. Así pues, lo mejor parece ser suministrarle otro argumento a la función indicando la base que deseamos emplear. Así llegamos a este otro programa.

```
 1 {
 2     escribir digitos separados
 3 }
 4 program digitos;
 5
 6 uses math;
 7
 8 function valordigito(n, pos, base: integer): integer;
 9 begin
10     result := (n div base**(pos-1)) mod base;
11 end;
12
13 const
14     Num: integer = 134;
15     Base: integer = 2;
16 begin
17     writeln(valordigito(Num, 3, Base));
18     writeln(valordigito(Num, 2, Base));
19     writeln(valordigito(Num, 1, Base));
```

```
20 end.
```

Ahora no sólo tenemos un subprograma que puede darnos el valor de un dígito para números en base 10. También podemos obtener los valores para dígitos en cualquier otra base.

Esto lo hemos conseguido **generalizando** el subprograma que teníamos. Vimos que teníamos uno que funcionaba en el caso particular de base 10. Utilizando un parámetro en lugar del literal 10 hemos conseguido algo mucho más útil. De hecho, hicimos lo mismo cuando nos dimos cuenta de que una única función bastaba y no era preciso utilizar una distinta para cada dígito.

si cuesta el mismo esfuerzo hacemos programas más generales

3.5.2. Valor numérico de un carácter

Queremos el dígito correspondiente a un carácter numérico. Por ejemplo, como parte de un programa que lee caracteres y devuelve números enteros.

Sabemos que los caracteres numéricos están en posiciones consecutivas en el código de caracteres Luego si obtenemos la posición del carácter y restamos la posición del carácter para el dígito "**´0´**" entonces tendremos el valor numérico deseado. Para obtener la posición, podemos utilizar **ord**.

```
function valornumerico(c: char): integer;
begin
    result := ord(c) - ord('0');
end;
```

3.5.3. Carácter para un valor numérico

Esta es similar. Basta sumar el valor a la posición del carácter "**´0´**" y obtener el carácter en dicha posición.

```
function cardigito(d: integer): char;
begin
    result := char(ord('0') + d);
end;
```

3.5.4. Ver si un carácter es un blanco.

Un blanco es un espacio en blanco o un tabulador. Podemos definir constantes para ambos caracteres. Como el problema consiste en ver si algo se cumple o no, deberíamos devolver un valor de verdad.

```
const
```

```
    Esp: char = ' ';
    Tab: char = '    '; {tabulador entre comillas}
function esblanco(c: char): boolean;
begin
    result := (c = Esp) or (c = Tab);
end;
```

3.5.5. Número anterior a uno dado módulo n

En ocasiones queremos contar de forma circular, de tal forma que tras el valor *n* no obtenemos $n + 1$, sino 0. Decimos que contamos módulo-n, en tal caso. Para obtener el número siguiente podríamos sumar 1 y luego utilizar el módulo. Para obtener el anterior lo que haremos será sumar*n* (que no cambia el v alor del número, módulo-n) y luego restar 1. Así obtenemos siempre un número entre 0 y$n - 1$.

```
function anteriormodn(num, n: integer): integer;
begin
    result := (num + n - 1) mod n;
end;
```

3.5.6. Comparar números reales

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

```
 1 {
 2     comparar reales
 3 }
 4 program compara;
 5
 6
 7 const
 8     A: real =  100.0 / 27.0;
 9 begin
10     writeln('Comparacion mal: ', A*27.0 = 100.0);
11 end.
```

diríamos que escribe **TRUE**. Pero si compilamos y ejecutamos, veremos que no.

```
unix% fpc compara.p
unix% compara
Comparacion mal: FALSE
```

En lugar de comparar los números reales con es mucho mejor ver si la diferencia, en valor absoluto, entre un número y aquel con el que lo queremos comparar es menor que un épsilon (un valor arbitrariamente pequeño). Veremos como programarlo. Demos por hecho que tenemos una forma de calcular el valor absoluto, **abs**, y una constante con el error que contemplamos para la comparación, **Eps** (llamada así por "epsilon'', que suele ser el nombre de una constante para un valor diminuto). Hecho esto, podemos realizar la función de comparación y resolver el problema. Si luego vemos que no existe **abs** en nuestro lenguaje de programación, pues la programamos y listo. Este es nuestro programa.

```
 1 {
 2     comparar reales
 3 }
 4 program compara;
 5
 6 const
 7     Eps: real = 0.000005;
 8
 9 function igualreal(a: real; b: real): boolean;
10 begin
11     igualreal := abs(a-b) < Eps;
12 end;
13
14 const
15     A: real =  100.0 / 27.0;
16 begin
17     writeln('Comparacion mal: ', A*27.0 = 100.0);
18     writeln('Comparacion bien: ', igualreal(A*27.0, 100.0));
19 end.
```

Si compilamos y ejecutamos el programa vemos esto:

```
unix% fpc compara.p
unix% compara
Comparacion mal: FALSE
Comparacion bien: TRUE
```

3.6. Pistas extra

Antes de continuar, veamos algunas cosas que te serán útiles. Recuerda que la forma de proceder es definir una función para efectuar el cálculo requerido. Para efectuar las pruebas de la función puede ser preciso definir constantes auxiliares para las pruebas.
Como has podido observar, para escribir por la salida estándar un dato se usa **write** o su variante **writeln**, que es igual que **write**, pero acaba la línea después de

escribir el dato que le pasamos como argumento. Como nota diremos que, en realidad, **`write`** acepta argumentos de distintos tipos. En base al tipo del argumento, actuará de una forma u otra (no hará lo mismo para escribir un entero que para escribir un real). A esto se le llama **polimorfismo**. Pascal lo usa para algunos de sus subprogramas predefinidos.

Para escribir reales, puedes utilizar la sintaxis de **`write`** que permite indicar cuántos caracteres quieres utilizar para escribir el valor de un argumento y cuántos decimales pra un argumento de tipo real. Por ejemplo,

```
writeln(3.12345 :0:2);
```

escribe

```
3.12
```

puesto que has indicado que se usen "**`0`**'' caracteres en total. Como no es posible, Pascal ha usado los que ha necesitado. Eso si, como se han pedido "**`2`**'' decimales, Pascal ha escrito justo esos.

Es muy importante seguir los pasos de resolución de problemas indicados anteriormente. Por ejemplo, si no sabemos cómo resolver nosotros el problema que queremos programar, difícilmente podremos programarlo. Primero hay que definir el problema, luego podemos pensar un algoritmo, luego programarlo y por último probarlo.

En este momento es normal recurrir a programas de ejemplo presentes en los apuntes y en los libros de texto para recordar cómo es la sintaxis del lenguaje. Si escribes los programas cada vez en lugar de cortar y pegar su código te será fácil recordar cómo hay que escribirlos.

3.7. Problemas

Escribe programas en Pascal que calculen cada uno de los siguientes problemas (Alguno de los enunciados corresponde a alguno de los problemas que ya hemos hecho, en tal caso hazlos tu de nuevo sin mirar cómo los hicimos antes).

1. Números de días de un año no bisiesto.
2. Volumen de una esfera.
3. rea de un rectángulo.
4. rea de un triángulo.
5. Volumen de un prisma de base triangular.
6. rea de un cuadrado.
7. Volumen de un prisma de base cuadrada.
8. Volumen de un prisma de base rectangular.
9. rea de una corona circular de radios interno y externo dados.
10. Volumen de un cilindro de radio dado al que se ha perforado un hueco vertical y cilíndrico de radio dado.
11. Primer dígito de la parte entera del número 14.5.

12. Raíz cuadrada de 2.
13. Carácter que se corresponde a la posición 87 en la tabla ASCII.
14. Factorial de 5.
15. Carácter siguiente a "**´X´**" .
16. Letras en el código ASCII entre las letras "**´A´**" y "**´N´**" .
17. Ver si un número entero tiene 5 dígitos.
18. Letra situada en la mitad del alfabeto mayúscula.
19. Media de los números 3, 5, 7 y 11.
20. Valor de la expresión

$$e^{2\pi}$$

21. Suma de los 10 primeros términos de la serie cuyos término general es

$$\left(1+\frac{1}{n}\right)^n$$

22. Convertir un ángulo de grados sexagesimales a radianes. Recuerda que hay 2π radianes en un círculo de 360^o.
23. Convertir de grados centígrados a farenheit una temperatura dada. Recuerda que

$$T_f = \frac{9}{5} T_c + 32$$

24. Ver si la expresión anterior (problema 21) es realmente el número *e*, cuyo valor es aproximadamente 2.71828.
25. Solución de la ecuación

$$x^2 + 15x - 3 = 0$$

26. Letra mayúscula correspondiente a la letra minúscula "**´j´**"
27. Energía potencial de un cuerpo de masa conocida a una altura dada.
28. Valor de verdad que indique si un carácter es una letra minúscula.
29. Ver si un carácter es una letra.
30. Ver si un carácter es un signo de puntación (considerados como tales los que no son letras ni dígitos).
31. Valor de verdad que indique si un número es par o impar.
32. Número de segundos desde el comienzo del día hasta las 15:30:27 horas.
33. Hora, minutos y segundos correspondientes a los 9322 segundos desde el comienzo del día.
34. Ver si un número es mayor que otro.
35. Número de espacios que hay que imprimir en una línea de longitud dada la la

izquierda de un texto para que dicho texto, de longitud también dada, aparezca centrado en dicha línea.

36. Ver si los números **a**, **b**, y **c** están ordenados de menor a mayor.
37. Carácter correspondiente al dígito **3**.
38. Dígito correspondiente al carácter .
39. Número medio de semanas en un mes.
40. Ver si el primer carácter tiene la posición cero.
41. Número siguiente a uno dado en aritmética modular con módulo 10.
42. Menor número entero disponible.
43. Número de caracteres presentes en el juego de caracteres en el sistema.
44. Ver si hay más caracteres que enteros o viceversa.
45. Ver si se cumple que el dividendo es igual a la suma del resto y del producto del divisor y cociente.
46. Expresión de verdad para determinar si un año es bisiesto. Son bisiestos los años múltiplo de 4 salvo que sean múltiplos de 100. Excepto por que los múltiplos de 400 también son bisiestos.
47. Ver si la suma del cuadrado de un seno de un ángulo dado mas la suma del cuadrado de su coseno es igual a 1.
48. Ver si un número está en el intervalo [x,y).
49. Ver si un número está en el intervalo [x,y].
50. Ver si un carácter es un blanco (espacio en blanco o tabulador).
51. Dadas las coordenadas **x** e **y** del punto de arriba a la izquierda y de abajo a la derecha de un rectángulo y dadas las coordenadas de un punto, ver si el punto está dentro o fuera del rectángulo. En este problema hay que utilizar aritmética de números enteros y suponer que el origen de coordenadas está arriba a la izquierda. El eje de coordenadas horizontal es el eje **x** y el vertical es el eje **y**
52. Ver si un número entero es un cuadrado perfecto (Esto es, es el cuadrado de algún número entero).
53. Ver si tres números son primos entre sí. Un número es primo con respecto a otro si no son divisibles entre sí.
54. Ver cuántos semitonos hay entre dos notas musicales.

Índice