

# Programación en Pascal. Expresiones

Miguel Ortuño  
Escuela de Ingeniería de Fuenlabrada  
Universidad Rey Juan Carlos

Marzo de 2024



© 2024 Miguel Angel Ortuño Pérez.  
Algunos derechos reservados. Este documento se distribuye bajo la  
licencia *Atribución-CompartirIgual 4.0 Internacional* de Creative  
Commons, disponible en  
<https://creativecommons.org/licenses/by-sa/4.0/deed.es>

- 1 Expresiones
  - Directivas para el compilador
  - Palabras reservadas
  - Comentarios
  - Identificadores
  - Tipos de Datos
  - Constantes
  - Escritura
  - Operadores
  - Codificación de caracteres
  - Casting
  - Representación de los números reales

# Directivas para el compilador

En este curso, siempre le pediremos al compilador que sea especialmente cuidadoso con los errores. Nos advertirá o prohibirá ciertas construcciones que en principio son legales, aunque peligrosas. Para ello añadimos la siguiente línea antes de la cabecera del programa:

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
```

- No te preocupes por su significado concreto, cópiala en todos tus programas

Ejemplo completo:

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
```

```
program holamundo;  
begin  
    writeln('Hola, mundo');  
end.
```

# Palabras reservadas

En prácticamente cualquier lenguaje de programación hay una serie de palabras que forman parte del propio lenguaje, no se pueden usar como identificadores.

Como referencia, incluimos aquí las de nuestro dialecto de Pascal (Object Pascal):

```
and array asm begin break case const constructor continue  
destructor div do downto else end false file for function  
goto if implementation in inline interface label mod nil  
not object of on operator or packed procedure program  
record repeat set shl shr string then to true type unit until  
uses var while with xor
```

# Comentarios

En cualquier lenguaje de programación hay *comentarios*. Se trata de texto que resulta útil para las personas, pero que el compilador ignorará. En nuestro dialecto de Pascal hay varias formas de insertar comentarios:

- Todo lo que se escriba entre llaves, es un comentario. Puede ser una línea o varias.

```
{Esto es un comentario  
de varias líneas}
```

- Todo lo que se escriba después de dos barras, hasta el final de esa línea, es un comentario. Por tanto es un comentario de una sola línea.

```
// Esto es un comentario de una línea.  
// Para añadir otro comentarios, escribo de nuevo dos barras
```

Los comentarios son importantes para la calidad de un programa.

- Los comentarios tienen que aportar información *relevante*.

```
writeln( ejemplo ); // Escribe en pantalla el valor de  
↪ 'ejemplo'
```

Este comentario es una obviedad, sobra.

- Un error típico de principiante (y de no tan principiante) es pensar que poner muchos comentarios ya hace que un programa esté bien documentado.
- La información que pueda extraerse directamente del código no debería repetirse en un comentario: se corre el riesgo de modificar el código y no actualizar el comentario.

Escribir comentarios informativos, relevantes y claros es muy importante. Aunque no es fácil para el principiante.

# Identificadores

Identificador: nombre para un elemento del programa (programa, función, procedimiento, constante, variable, etc).

- Normalmente definido por el programador (o por el programador de un *módulo*<sup>1</sup>).
- En Pascal solo podemos usar letras inglesas para los identificadores.
  - Esto no suele ser un problema, cualquier programa medianamente serio estará escrito en inglés (identificadores y comentarios). Otros idiomas como el español se usan solo en el interface de usuario, si procede.

---

<sup>1</sup>Módulo: fichero que contiene código para ser usado desde un programa en otro fichero distinto. En otros lenguajes a los módulos se les llama librerías



- El lenguaje Pascal no distingue mayúsculas de minúsculas. Apellidos, APELLIDOS y APELLIDOS resulta equivalente.
  - La mayoría de los lenguajes de programación sí distinguen mayúsculas de minúsculas.
- Es importante elegir buenos identificadores, que indiquen claramente qué nombran. Un identificador ambiguo o abiertamente incorrecto es un error de claridad en el programa. No provoca errores lógicos por si mismo, pero induce al programador a crearlos.
  - El sulfato de sodio hay que etiquetarlo como *sulfato de sodio*, no como *sulfato*. Y mucho menos como *sulfato de cloro*. De lo contrario ... <https://youtu.be/QNTZbJSQVis>
- Debemos esforzarnos en elegir buenos identificadores, aunque esto no es fácil para el principiante. Leer atentamente ejemplos de código de calidad ayuda a adquirir esta habilidad.

# Ámbito de un identificador

- Algunos identificadores no se pueden repetir, tienen que ser únicos en todo el programa. Son de *ámbito global*.
- Otros identificadores sí se pueden repetir, no importa si aparece el mismo en dos lugares distintos del programa, estarán nombrando cosas distintas. Son de *ámbito local*.

En otras palabras:

- Un identificador tiene que ser único en su ámbito. Si su ámbito es global, tiene que ser único en todo el programa. Si su ámbito es local, tiene que ser único en cierta parte del programa, pero puede repetirse en otro lugar del programa.
- *Ámbito* es el lugar de un programa donde se puede usar un identificador.

Cada lenguaje de programación tiene sus propias reglas sobre el ámbito de los identificadores, pero suelen ser muy parecidas: las mismas que Pascal.

Por ejemplo, en Pascal, el nombre de programa es de ámbito de global. El identificador que nombra a un programa no puede repetirse para nombrar otra cosa distinta en otro lugar del programa.

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ejemplo;  
const  
    Ejemplo = 3.0; // ;MAL! identificador repetido  
begin  
    writeln( Ejemplo );  
end.
```

Si intentamos compilar el ejemplo anterior, obtendremos un error.

```
koji@mazinge:~/fpi/tema02$ fpc -gl ejemplo.pas
```

```
Free Pascal Compiler version 3.0.4+dfsg-23 [2019/11/25] for x86_64
```

```
Copyright (c) 1993-2017 by Florian Klaempfl and others
```

```
Target OS: Linux for x86-64
```

```
Compiling ejemplo.pas
```

```
ejemplo.pas(5,18) Error: Duplicate identifier "Ejemplo"
```

```
ejemplo.pas(10) Fatal: There were 1 errors compiling module, stopping
```

```
Fatal: Compilation aborted
```

```
Error: /usr/bin/ppcx64 returned an error exitcode
```

# Tipos de datos

En Pascal manipulamos datos. Son de 5 tipos:

- Integer. Números enteros.
- Real. Números reales.
- Char. Carácteres.  
'a' es un tipo char. También ' ' y '0', que no debemos confundir con el número 0.
- String. Cadenas de texto.
- Boolean. Valores booleanos.  
Solo puede tomar dos valores: TRUE o FALSE.

Para un ordenador es muy distinto el número real 3, el número entero 3, el carácter '3', o la cadena '3'.

- El problema es que para las personas no familiarizadas con la programación, no existe esta diferencia.
- Además, una vez escrito en pantalla o en papel, es imposible distinguirlo (en el código fuente sí).

A veces representamos (en pantalla, papel o el código fuente) un número real sin decimales con un cero tras la coma para distinguirlo.

Ejemplo: el real 3, a veces se escribe como 3.0 (pero no siempre). Así queda claro que no es el entero 3.

# Caracteres y cadenas

En Pascal los valores de los caracteres y de las cadenas se escriben con una comilla recta al principio y otra al final.

'esto es una cadena'

- Hay lenguajes que usan la comilla doble.
- Hay lenguajes que permiten usar tanto la comilla doble como la recta.

Observa que en el teclado español, la comilla recta es la tecla a la derecha de la tecla 0.

- No la confundas con la comilla invertida, la tecla a la derecha de la tecla p.
- En ciertas tipografías, lamentablemente la comilla recta no se representa recta, lo que induce a confusión.

# Declaración de constantes

Una constante es una entidad (una *caja*) que contiene un dato, que no cambiará durante la ejecución del programa.

- Nos referiremos a ella con un identificador. Un convenio habitual, que seguiremos en este curso, es escribirlas empezando por letra mayúscula.

Para usar una constante:

- 1 Podemos declararla, indicando su tipo.  
Nombre de la constante, dos puntos, tipo de dato, punto y coma.
- 2 La definimos, indicamos su valor.  
Nombre de la constante, igual, valor, punto y coma.

El primer punto es opcional. Casi siempre se puede elegir declarar o no declarar, va en gustos. Aquí recomendamos que no declares las constantes.



Ejemplo de declaración y definición:

```
const  
  E: real ;  
  E = 2.71828182845904;
```

Si declaramos y definimos, es recomendable declarar y definir en la misma línea:

```
const  
  E: real = 2.71828182845904;
```

Aquí recomendamos definir sin declarar. Esto es, indicar el valor pero no el tipo<sup>2</sup>.

```
const
```

```
  E = 2.71828182845904;
```

La definición, con o sin declaración, ha de hacerse:

- Dentro de un bloque, en la parte declarativa , después de la palabra reservada `const` y antes de la lista de sentencias (begin end).

Observaciones:

- Después de `const` no va un punto y coma.

---

<sup>2</sup>En el tema 8 veremos que el tamaño de los arrays es necesario (y no optativo) definirlos así, sin declarar el tipo

Las constantes pueden declararse

- Al principio del programa.  
Serán constantes globales, visibles en todo el programa. Deben usarse lo mínimo posible, solo para valores *universales*, que no cambien fácilmente: p.e. el número Pi, el radio de la tierra, el tamaño de un campo en un fichero estandarizado, etc<sup>3</sup>.
- Al principio de un subprograma.  
Típicamente, al principio del cuerpo del programa principal. Solo serán visibles en este subprograma. Aquí podemos definir p.e. datos concretos de nuestro programa.

Si tenías nociones de programación, echarás de menos las variables. Por motivos didácticos, en este curso las veremos en el tema 5. No las uses hasta entonces.

---

<sup>3</sup>Suponiendo que todo esto sea constante en el ámbito de nuestro problema, en ciertos escenarios todos estos valores podrían ser cambiantes.

# Escritura en pantalla

- El procedimiento `write` escribe en la consola (la pantalla) los argumentos que recibe.
- El procedimiento `writeln` escribe en la consola los argumentos que recibe, y a continuación, un salto de línea.
- Este procedimiento escribe los valores reales en notación científica. Para usar notación decimal, añade a continuación `:0:n`, donde
  - El 0 significa que el número puede ocupar todo el espacio que necesite.
  - `n` representa el número de decimales.

Ejemplo: `write(Tiempo_segundos:0:1)`

- Esta es una de las pocas rarezas de Pascal, no es habitual encontrarlo en otros lenguajes.

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program talla01;  
  
const  
  Talla = 'xl';  
  Color = 'azul';  
begin  
  writeln('Talla:');  
  writeln(Talla);  
  writeln('Color:');  
  writeln(Color);  
end.
```

Con el procedimiento *writeln*, cada argumento se escribe en una línea distinta, este es el resultado:

```
Talla:  
xl  
Color:  
azul
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program talla02;  
  
const  
  Talla = 'xl';  
  Color = 'azul';  
begin  
  write('Talla:');  
  write(Talla);  
  write('Color:');  
  write(Color);  
end.
```

Con el procedimiento *write*, todos los argumentos se escriben la misma línea, sin espacios por medio, este es el resultado:

Talla:xlColor:azul

Normalmente usaremos una combinación de *write* y *writeln*, como en el siguiente ejemplo:

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program talla03;  
  
const  
    talla = 'xl';  
    color = 'azul';  
begin  
    write('Talla:');  
    writeln(talla);  
    write('Color:');  
    writeln(color);  
end.
```

Resultado:

```
Talla:xl  
Color:azul
```

# Reales en notación científica

Por omisión, nuestro dialecto de Pascal escribe todos los números reales en notación científica

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program descuento01;  
  
const  
    Precio = 12.50;  
    Porcentaje_descuento = 10.0;  
begin  
    write('Precio: ');  
    write(Precio);  
    write(' Descuento: ');  
    writeln(Precio * 0.01 * Porcentaje_descuento );  
    write('Precio final: ');  
    writeln(Precio * (1 - 0.01 * Porcentaje_descuento));  
end.
```

Resultado:

```
Precio: 1.2500000000000000E+001 Descuento: 1.2500000000000000E+0000  
Precio final: 1.1250000000000000E+0001
```



# Reales en notación convencional

Para escribir los números reales en notación convencional, añadimos `:0:n`, donde  $n$  es el número de decimales deseados.

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program descuento02;  
  
const  
    Precio = 12.50;  
    Porcentaje_descuento = 10.0;  
begin  
    write('Precio: ');  
    write(Precio:0:2);  
    write(' Descuento: ');  
    writeln(Precio * 0.01 * Porcentaje_descuento:0:2 );  
    write('Precio final: ');  
    writeln(Precio * (1 - 0.01 * Porcentaje_descuento):0:2 );  
end.
```

Resultado:

Precio: 12.50 Descuento: 1.25

Precio final: 11.25

# Definición de reales en notación científica

Naturalmente, también podemos emplear la notación científica para definir número reales.

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program cientifica;
const
    // Podemos escribir 'e' o 'E'
    Billon_europeo = 1e+12;
    Billon_americano = 1E+9;
    Cte_Planck = 6.62607015E-34;

begin
    writeln( 'Notación científica' );
    writeln( 'Billón europeo:', Billon_europeo );
    writeln( 'Billón americano:', Billon_americano );
    writeln( 'Constante de Planck', Cte_Planck );
    writeln();
    writeln( 'Notación tradicional' );
    writeln( 'Billón europeo:', Billon_europeo:0:0 );
    writeln( 'Billón americano:', Billon_americano:0:0 );
    writeln( 'Constante de Planck:', Cte_Planck:0:36 );
end.
```

## Resultado del programa anterior:

### Notación científica

Billón europeo: 1.00000000000000000000E+0012

Billón americano: 1.000000000E+09

Constante de Planck 6.626070150000000000011E-0034

### Notación tradicional

Billón europeo:1000000000000

Billón americano:1000000000

Constante de Planck:0.00663

Atención: para usar `:0:n` es necesario que el argumento del `write` sea real, no puede ser entero.

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program error_write;

const
  X = 3.0;
  Y = 7;

begin
  write('X:');
  writeln(X:0:2);
  write('Y:');
  writeln(Y:0:2); //;;Mal!! La constante Y es entera
end.
```

Compiling error\_write.pas

error\_write.pas(12,18) Error: Illegal use of ':'

error\_write.pas(14) Fatal: There were 1 errors compiling module, stopping

# Expresiones: otro ejemplo

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program area_triangulo;  
  
const  
    Base = 12.43;  
    Altura = 5.91;  
  
begin  
    write('Base:');  
    write(Base:0:3);  
    write(' altura: ');  
    writeln(Altura:0:3);  
    write('Área del triángulo: ');  
    writeln(Base * Altura * 0.5:0:3);  
end.
```

Resultado de la ejecución:

```
Base:12.430 altura: 5.910  
Área del triángulo: 36.731
```

En un mismo write o writeln se pueden escribir varios argumentos, separados por comas.

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program area_triangulo;  
  
const  
    Base = 12.43;  
    Altura = 5.91;  
  
begin  
    write('Base:', Base:0:3);  
    writeln(' altura: ', Altura:0:3);  
    writeln('Área del triángulo: ', Base * Altura * 0.5:0:3);  
end.
```

Resultado:

```
Base:12.430 altura: 5.910  
Área del triángulo: 36.731
```

## Sugerencia:

- Mientras seas principiante, escribe un solo argumento en cada `write/writeln`.
- Cuando tengas más soltura, puedes escribir dos argumento en cada `write/writeln`, pero evita escribir más de dos.

## ¿Por qué?

- Para evitar errores (comillas mal colocadas, paréntesis, etc).
- Para que los mensajes de error del compilador sean más claros.

# Operadores

Un operador es un símbolo o una palabra reservada que indica que se debe realizar una operación matemática o lógica sobre unos *operandos* para devolver un resultado.

- Los operandos son expresiones (valores) de entrada, en Pascal la mayoría de los operadores tienen 2 operandos, algunos tienen 1.

Ejemplo:

- $5 + 3$

El operador  $+$  tiene dos operandos: el 5 y el  $3^4$  y devuelve como resultado su suma.

---

<sup>4</sup>a simple vista no podemos saber si son enteros o reales, pero en este caso no importa, el compilador hará la conversión si hace falta



Los operadores están muy vinculados a los tipos de datos, cada operando solo puede recibir ciertos tipos concretos de datos, para devolver cierto tipo de datos.

### Ejemplo

El operador `div` es la división entera. Sus operandos han de ser números enteros. En otro caso, el compilador produce un error.

Uso correcto:

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ej_div;  
begin  
    writeln( 5 div 2);    // Escribe 2  
end.
```

## Uso incorrecto:

```
writeln( 5 div 2.0);
```

```
koji@mazinger:~/pascal$ fpc ej_div.p
Free Pascal Compiler version 3.0.0+dfsg-2 [2016/01/28] for x86_64
Copyright (c) 1993-2015 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling ej06.p
ej06.p(4,16) Error: Operator is not overloaded: "ShortInt" div "Single"
ej06.p(8) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

# Otro ejemplo incorrecto

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program tipos_mal;

const
  X = 3;
  Y = '4';
begin
  write( 'X vale ');
  writeln( X );

  write( 'Y vale ');
  writeln( Y );

  write( 'La suma vale ');
  write( X + Y ); // ¡¡MAL!! La constante Y no es numérica
end.
```

Compiling tipos\_mal.pas

tipos\_mal.pas(15,14) Error: Operator is not overloaded: "LongInt" + "Char"

tipos\_mal.pas(17) Fatal: There were 1 errors compiling module, stopping

Fatal: Compilation aborted

Error: /usr/bin/ppcx64 returned an error exitcode

# Potencias

Para poder usar el operador de exponenciación, es necesario añadir la cláusula `uses math;` en la cabecera, esto añade el módulo matemático.

Ejemplo:  $2,1^{3,1}$

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program ej_potencias;  
uses math;  
begin  
    writeln( 2.1 ** 3.1); // Escribe 9.97423999265870760145E+0000  
end.
```

# Codificación de caracteres

Los ordenadores almacenan toda la información, incluyendo los textos, como números (binarios).

- Cada carácter se asocia con un código numérico. A esto se le denomina *codificación de caracteres*.

[https://es.wikipedia.org/wiki/Codificaci%C3%B3n\\_de\\_caracteres](https://es.wikipedia.org/wiki/Codificaci%C3%B3n_de_caracteres)

- En la década de 1960 se populariza el código ASCII, que asocia cada letra (inglesa) con un número.
- Ha habido diferentes formas de representar las letras no inglesas.
- Desde la década de 2010 (aprox.) lo más habitual es emplear la norma *Unicode* codificada en *UTF-8*.
  - Es un superconjunto de ASCII: esto es, las letras inglesas se codifican exactamente igual que en ASCII.

# Casting

- Pascal es *fuertemente tipado*, esto significa que en general no se pueden mezclar tipos de datos, hay que convertirlos antes para que sean homogéneos.  
Convertir un dato de un tipo a otro se llama *ahormado*. En español solemos emplear la palabra inglesa: *casting*.
- Algunas conversiones de tipos las hace el compilador automáticamente.

# Operadores para enteros y reales

Hay operadores numéricos que se pueden usar con enteros o con reales, indistintamente, incluso mezclando los tipos.

\*\* Exponenciación  
+ - \* /  
- (operador unario de cambio de signo)

Realmente lo que sucede es que si un operando es entero y el otro real, el compilador convierta automáticamente el entero en real.

Ejemplo:  $2 + 2.0$  internamente se convierte en  $2.0 + 2.0$

# Operadores solo para enteros

Otros operadores solo admiten operandos enteros.

- `div`

El operador *div* devuelve la división entera: la división de dos operandos enteros.

Ejemplo: `5 div 2 = 2`

- `mod`

El operador *mod* devuelve el módulo de la división entera: el resto <sup>5</sup>.

Ejemplo: `5 mod 2 = 1`

Si el programador se equivoca e intenta usarlos con operandos reales, el compilador dará un error. Ejemplo:

```
6.12 div 0.02    // ¡MAL!  
4.0 mod 2.0      // ¡MAL!  
4 mod 2.0        // ¡MAL!
```

---

<sup>5</sup>Aquí 'módulo' es sinónimo de 'resto'. Nada que ver con 'módulo' con el significado de 'librería'.



El programador puede hacer ciertas conversiones de tipos explícitamente.

- Un entero se puede convertir en real.

```
real(3)
```

- Un carácter se puede convertir en entero (obteniendo el código ASCII correspondiente).

```
integer('a')
```

- Un entero se puede convertir en carácter (obtenemos el carácter correspondiente a ese código ASCII).

```
char(97)
```

Código ASCII:

<https://es.wikipedia.org/wiki/ASCII>

Un número real no se puede ahormar directamente a entero,  
Pero disponemos de las funciones predefinidas  
`trunc()` y `round()`  
que reciben un número real y devuelven un entero, truncado sus  
decimales o redondeando.

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program round_trunc;  
const  
    X= 4.7;  
    Y= 4.5;  
    Z= 4.5001;  
begin  
    writeln( round(X) ); // 5  
    writeln( trunc(X) ); // 4  
    writeln( round(Y) ); // 4  
    writeln( round(Z) ); // 5  
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
```

```
program casting;
```

```
const
```

```
  X = 'a';
```

```
  Y = 98;
```

```
  Z = 65.7;
```

```
begin
```

```
  writeln( integer(X) ); // Escribe 97
```

```
  writeln( char(Y) ); // Escribe 'b'
```

```
  writeln( real(Y) ); // Escribe 9.800000000000000E+001
```

```
  writeln( real(Y):0:2 ); // Escribe 98.00
```

```
  { writeln( integer(Z) ); // ¡Esto es ilegal! }
```

```
  writeln( round(Z) ); // Escribe 66
```

```
  writeln( trunc(Z) ); // Escribe 65
```

```
  writeln( char( trunc(Z) ) ); // Escribe 'A'
```

```
end.
```

# Operadores de comparación

Sus argumentos pueden ser enteros o reales. Devuelven un booleano.

= Igual  
<> Distinto  
< Menor  
<= Menor o igual  
> Mayor  
>= Mayor o igual

Un error frecuente es confundir el operador de comparación de igualdad = con el operador de asignación := <sup>6</sup>.

---

<sup>6</sup>O con el operador de comparación de igualdad en C y derivados ==, o con el operador de asignación en C y derivados, =.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}  
program compara;  
  
const  
    A = 50;  
    B = 100;  
begin  
    writeln(A = B); // FALSE  
    writeln(A = A); // TRUE  
    writeln(A <> B); // TRUE  
    writeln(A > B); // FALSE  
    writeln(A <= B); // TRUE  
    writeln((A+A) >= B); // TRUE  
end.
```

Es materia de los temas 3 y 4, pero adelantamos aquí estos ejemplos.

- programa `ej_comparacion_v1`;  
La función `posible_matricula` devuelve TRUE si la nota es 10, devuelve FALSE en otro caso.
- programa `ej_comparacion_v2`;  
La función `posible_matricula` devuelve la expresión `nota=10`.  
Valdrá TRUE si la nota es 10, FALSE en otro caso.

Ambos programas devuelven lo mismo.

- El principiante tiende a utilizar la primera forma, que es redundante pero no es incorrecta, al principiante le resulta más claro y la claridad es conveniente.
- El programador con más experiencia preferirá la segunda.

```
={$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program ej_comparacion_v1;

function posible_matricula(nota: real):boolean;
begin
    if nota = 10
    then
        result := TRUE
    else
        result := FALSE
    end;

const
    nota_ejemplo: real = 9.5;
begin
    writeln( posible_matricula(nota_ejemplo))
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program ej_comparacion_v2;  
  
function posible_matricula(nota: real):boolean;  
begin  
    result := (nota = 10); // Devuelve un booleano  
end;  
  
const  
    nota_ejemplo: real = 9.5;  
begin  
    writeln( posible_matricula(nota_ejemplo))  
end.
```



# Números reales: separador decimal

Para separar la parte decimal de la parte fraccionaria de un número real...

- En los países anglosajones se usa tradicionalmente el punto.
- En países latinoamericanos del norte de América (México, caribe) se usa tradicionalmente la coma.
- La tradición española coincide con la franco belga: la coma.
- En el año 2010, la Real Academia Española cambia de criterio y recomienda usar el punto y no la coma, aunque ambas opciones siguen siendo correctas.
- Los lenguajes de programación siempre usan el punto.
- Las hojas de cálculo usan el punto o la coma según como esté configurado el idioma en esta aplicación.

En resumen: es un asunto al que hay que prestar atención.

Hay una costumbre tradicional que actualmente desaconsejan todas las normativas y recomendaciones internacionales.

- Usar la coma para separar grupos de tres dígitos cuando el punto separe decimales.  
*3,000.150 tres mil punto ciento cincuenta ¡Mal!*
- Usar el punto para separar grupos de tres dígitos cuando la coma separe decimales.  
*3.000,150 tres mil coma ciento cincuenta ¡Mal!*

Lo que se recomienda en este caso es usar espacios, salvo que puedan inducir a confusión.

*3 000 000 tres millones*

# Representación de los números reales

La representación convencional es poco adecuada para ciertos números reales. En matemáticas en general (no solo en programación) se usa entonces la *notación científica*.

- Los números reales tienen un número de decimales potencialmente infinito.
- Pueden ser muy grandes, con un número de dígitos muy elevado.
- Pueden ser muy pequeños, con muchos ceros tras la coma.

Un número real  $r$  en notación científica se representa:

$$r = c \times b^e$$

$c$  es el *coeficiente*,  $b$  es la base y  $e$  es el *exponente*.

Ejemplos:  $6,022 \times 10^{23}$        $1 \times 10^{-8}$

Al coeficiente comúnmente se le llama *mantisa*, aunque en rigor esta denominación es incorrecta <sup>7</sup>.

---

<sup>7</sup>mantisa es la parte decimal de un logaritmo.

# Notación de ingeniería

La notación de ingeniería es un caso particular de notación científica, donde el exponente ha de ser múltiplo de 3.

Ejemplo:  $10,3 \times 10^3$

Cada uno de estos exponentes tiene un nombre, todo ello facilita la lectura.

Factor	Prefijo	Símbolo
$10^{-15}$	femto	f
$10^{-12}$	pico	p
$10^{-9}$	nano	n
$10^{-6}$	micro	$\mu$
$10^{-3}$	mili	m
$10^3$	kilo	K
$10^6$	mega	M
$10^9$	giga	G
$10^{12}$	tera	T
$10^{15}$	peta	P

# Coma flotante

Los ordenadores usan un caso particular de notación científica, la *representación de coma flotante* (o *coma flotante*, *floating point*) siguiendo el estándar IEEE 754 (año 1985).

Incluye 5 formatos básicos: tres de base dos y dos de base diez. Los más usados son:

- *binary32* (binario, simple precisión)  
Se corresponde con el tipo *real* de Free Pascal. Es el único que usaremos en esta asignatura, lo habitual en entornos no especialmente exigentes.
- *binary64* (binario, doble precisión).  
Se corresponde con el tipo *double* de Free Pascal.

Cada formato ocupa diferente espacio en memoria, con diferente tamaño del coeficiente y del exponente. Por tanto, cada formato tendrá distinto.

- Rango: valores mínimo y máximo representables.
- Resolución numérica: valor mínimo, no negativo, no nulo.
- Cifras significativas: dígitos que aporten información.

[https://es.wikipedia.org/wiki/Cifras\\_significativas](https://es.wikipedia.org/wiki/Cifras_significativas)

Puedes consultar las características de cada formato en [https://es.wikipedia.org/wiki/IEEE\\_754](https://es.wikipedia.org/wiki/IEEE_754)

# Errores de conversión

Tomemos por ejemplo el formato real más habitual de muchos lenguajes: IEEE 754 *binary32*. Admite un exponente máximo de 8 bits, lo que equivale a 38.23 dígitos binarios.

- Podría pensarse que esto permite un rango muy grande y una precisión muy alta, que sería raro que nuestro problema requiera de más precisión.
- Pero hay otro aspecto importante a considerar: los errores de conversión binario / decimal.
  - Los humanos casi siempre usamos notación decimal, los ordenadores siempre usan internamente notación binaria, esto obliga a continuas conversiones que provocan errores.
  - Usando *binary32*, es fácil encontrar errores a partir del quinto decimal, como puedes ver en este conversor online:  
<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

Por todo ello, en aquellas situaciones donde necesitemos una precisión superior a unos pocos decimales, tendremos que prestar atención a los tipos de datos reales de nuestros programas.

- Esto no solo depende del lenguaje, también del dialecto, del compilador e incluso de la arquitectura (CPU) concreta de cada caso.
- Por ejemplo en Free Pascal los tipos de real soportados son *real*, *single*, *double*, *extended*, *comp*, *currency*.  
<https://www.freepascal.org/docs-html/ref/refsu5.html>



# Operadores booleanos

- `op1 and op2`  
Devuelve TRUE cuando ambos operandos son ciertos.  
Devuelve FALSE en otro caso.
- `op1 or op2`  
Devuelve TRUE cuando un operando es cierto o cuando dos operandos son ciertos.  
Devuelve FALSE en otro caso.
- `not op`  
Devuelve TRUE cuando el operando es FALSE.  
Devuelve FALSE cuando el operando es TRUE.
- `op1 xor op2`  
Devuelve TRUE cuando un operando es TRUE y otro es FALSE.  
Devuelve FALSE en otro caso.  
equivale a  
 $(op1 \text{ and } (\text{not } op2)) \text{ or } ((\text{not } op1) \text{ and } op2)$

# Expresiones booleanas

- Ya estás familiarizado con las expresiones numéricas, que combinan operandos numéricos con operadores numéricos. P.e  
`1.23 + (2.4 * 12)`

- En programación se usan mucho, además, las expresiones booleanas. Los operandos son booleanos y, por supuesto, también los operadores.

```
FALSE or not (TRUE and FALSE)  
Diabetico and not Menor_edad
```

- Los operadores booleanos solo admiten operandos booleanos.

```
Diabetico and 20 // ¡¡Mal!!
```

- Los operadores de comparación aceptan enteros o reales como operandos, y devuelven un booleano, con el que ya podemos construir expresiones booleanas.

```
Diabetico and (Edad >= 18)
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program franja;

const
    Cota_inferior = -100;
    Cota_superior = 100;
    Valor = 20;
begin
    // writeln( Cota_inferior <= Valor <= Cota_superior);
    // ¡MAL! En matemáticas escribimos a<b<c. Aquí no podemos

    // writeln( Cota_inferior <= Valor and Valor <= Cota_superior);
    // ¡MAL! and es un operador booleano, pero Valor y Valor son
    // enteros. Es necesario usar paréntesis

    writeln( (Cota_inferior <= Valor) and (Valor <= Cota_superior));
    // Correcto
end.
```

# Lógica proposicional

Doble negación:

- $\neg\neg p \Leftrightarrow p$
- No es cierto que no llueva  $\Leftrightarrow$  Llueve

Ambas expresiones son lógicamente equivalentes, aunque la segunda es más clara.

En lenguaje natural, es habitual usar expresiones del tipo *si... entonces*.

- *Si me avisas, entonces llevo más dinero.*
- *Si suspendes las prácticas, entonces suspendes la asignatura.*

En la especificación de un algoritmo hay que tener mucho cuidado, porque en rigor, con esta estructura estamos diciendo qué sucede si se cumple la condición, pero no estamos diciendo qué pasa si no se cumple.

- En algunos casos, posiblemente los humanos supongan que si no se cumple la condición, no se cumple la consecuencia.  
*Si no me avisas, entonces no llevo más dinero.*
- En otros, posiblemente no haremos esa suposición.  
*Si apruebas las prácticas, ya veremos, dependes de los exámenes.*

Estas imprecisiones propias del lenguaje natural no son admisibles en la especificación de un algoritmo, es importante dejar claro qué pasa si la condición es falsa: si estoy diciendo algo para ese caso o si no estoy diciendo nada.

*Si (condición) entonces (consecuencia)*

¿Qué pasa si no se cumple la condición?

- Caso 1. Equivalencia.

Entonces tampoco se cumple la consecuencia.

- Caso 2. Implicación.

Entonces no digo nada sobre consecuencia, puede que se cumpla, puede que no.

En lenguaje natural tenemos estas dos posibilidades, en lenguaje formal, solo la segunda.

## Caso 1. Equivalencia

- *Si me avisas, entonces llevo más dinero. Y si no, no.*

Esto se convierte en una equivalencia lógica. El aviso equivale a llevar más dinero.

$$\text{aviso} \implies \text{mas\_dinero} \wedge \neg \text{aviso} \implies \neg \text{mas\_dinero}$$

$$\text{aviso} \Leftrightarrow \text{mas\_dinero}$$

- $p \implies q \wedge \neg p \implies \neg q$

$$p \Leftrightarrow q$$

## Caso 2. Implicación.

- *Si suspendes las prácticas, entonces suspendes la asignatura.  
Y si apruebas las prácticas, ya veremos.*  
*suspender\_practicas  $\implies$  suspender\_asignatura*  
(Y ya está, en un entorno formal está claro que no estoy haciendo ninguna afirmación si no se da la condición).
- $p \implies q$



Ojo con sacar conclusiones erróneas

- $p \implies q$

¿Equivale a ?

$$\neg p \implies \neg q$$

¡No!

Con otras palabras, es lo mismo que acabamos de ver. De la afirmación *si me avisas, entonces llevo más dinero*, no se puede deducir que si no me avisas, no lo llevo. Es necesario indicarlo explícitamente (si procede).

# Transposición de la implicación

La conclusión que sí podemos extraer es

- $p \implies q \Leftrightarrow \neg q \implies \neg p$
- *ser asturiano implica ser español*  
equivale a  
*no ser español implica no ser asturiano.*

Otro ejemplo

- Si he venido es porque no lo sabía.
- Si lo se no vengo.

Un poco más claro, en presente:

- Si voy es porque no lo se.
- Si lo se, no voy.

# Leyes de De Morgan

Las leyes de De Morgan<sup>8</sup> también permiten generar expresiones booleanas equivalentes desde el punto de vista lógico.

- `not (a and b)`  
equivale a  
`(not a) or (not b)`
- `not (a or b)`  
equivale a  
`(not a) and (not b)`

Ejemplos:

`not (joven and rico) ⇔ not joven or not rico`

`not (anciano or niño) ⇔ not anciano and not niño`

---

<sup>8</sup>No es una errata, el nombre de su descubridor es Augustus De Morgan

Aplicando la doble negación y las leyes de De Morgan, podemos escribir las expresiones booleanas de formas distintas, que

- Desde el punto de vista lógico y matemático, serán equivalentes.
- Considerando la claridad para el humano, no serán equivalentes. Las personas entendemos mejor la *lógica positiva* (afirmaciones sin negaciones) que la *lógica negativa* (afirmaciones con negaciones).

En programación, normalmente lo más importante es el programador, presente o futuro. En general deberemos usar la expresión equivalente con menos negaciones.

Aplicando la transposición de la implicación, podemos mejorar la claridad de las implicaciones en lógica booleana.

# Ejemplos

- Niega la siguiente expresión y simplifica el resultado

$$p = \text{not } q \text{ and } (\text{not } r \text{ or } s)$$

Solución:

$$\text{not } p = \text{not } (\text{not } q \text{ and } (\text{not } r \text{ or } s))$$

$$\text{not } p = q \text{ or not } (\text{not } r \text{ or } s)$$

$$\text{not } p = q \text{ or } (r \text{ and not } s)$$

- Niega la siguiente expresión y simplifica el resultado

$$p = (a \geq 65) \text{ or } (a \leq 16) \text{ or } q$$

Solución:

$$\text{not } p = (a < 65) \text{ and } (a > 16) \text{ and not } q$$

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program diabetes;

const
    Diabetico = TRUE;
    Menor_edad = FALSE;
    Edad = 20;
begin
    writeln(FALSE or not (TRUE and FALSE)); // Escribe TRUE
    writeln(Diabetico and not Menor_edad); // Escribe TRUE

    { writeln(Diabetico and not Edad); ;; Esto es un error !! }
    { writeln(Diabetico and not Edad < 18); ;; Esto es un error !! }

    writeln(Diabetico and not (Edad < 18)); // Escribe TRUE
    writeln(Diabetico and (Edad >= 18)); // Escribe TRUE
end.
```

# Ejemplo: años bisiestos

- Descripción en lenguaje natural:

Los años múltiplos de 4 son bisiestos.

Excepción: los múltiplos de 100, que no lo son.

Excepción a la excepción: los múltiplos de 400 sí lo son.

## En otras palabras

Los años múltiplos de 4 son bisiestos.

Excepción: múltiplo de 100 y no múltiplo de 400.

## En otras palabras

Un año es bisiesto si es múltiplo de 4 y no se cumple que:  
es múltiplo de 100 y no es múltiplo de 400.

- Aplicando De Morgan, llegamos a la descripción algorítmica

Un año es bisiesto si es múltiplo de 4 y  
(no es múltiplo de 100 o es múltiplo de 400).

## Implementación en Pascal:

```
(A mod 4 = 0 ) and (not (A mod 100 = 0) or (A mod 400 = 0) )
```

## Implementación en Pascal, un poco más clara:

```
(A mod 4 = 0 ) and ( (A mod 100 <> 0) or (A mod 400 = 0) )
```



```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program bisiesto ;  
const  
    Anyo = 1940;  
  
begin  
    write(Anyo, ' es bisiesto:');  
    writeln(  
        (Anyo mod 4 = 0 )  
        and  
        ( (Anyo mod 100 <> 0) or (Anyo mod 400 = 0 ))  
    );  
end.
```

# Ejercicio

- 1 Escribe en una hoja de papel un par de expresiones booleanas parecidas a las de la pg 63.
  - Una equivalencia, con la constante  $p$  a la izquierda de la igualdad y las constantes  $q$ ,  $r$  y  $s$  a la derecha. Con los operadores and, or, algún not y algún paréntesis.
  - Otra equivalencia con la constante  $p$  a la izquierda, valores numéricos y constantes a la derecha. Con operadores de comparación, and y or.
  - Usa la notación de Pascal (and, or, not), no uses notación lógica ( $\wedge, \vee, \neg$ ).
- 2 Escribe en otra hoja las expresiones booleanas equivalentes, negando ambos lados de la igualdad.
- 3 Entrega la primera hoja a un compañero para haga lo mismo. Resuelve tú su ejercicio. Comparad las soluciones para comprobar que sean iguales, corregid el problema si hay errores.

# Precedencia de operadores

En matemáticas normalmente podemos distribuir los operandos entre varias líneas, haciendo cosas como  $\frac{4+2}{1+1} = 3$ .

- En casi todos los lenguajes de programación, nos vemos obligados a usar una sola línea.
- Si intentamos escribir la expresión anterior como  $4 + 2/1 + 1$  estaremos cometiendo un error, porque el compilador lo interpreta como  $4 + \frac{2}{1} + 1 = 7$ .

Los operadores tienen una *reglas de precedencia*:

- Los operadores se evalúan de mayor precedencia a menor precedencia.
- A igualdad de precedencia, se evalúa de izquierda a derecha.

Precedencia en Pascal, de mayor a menor:

```
** not - (cambio signo)
* / div mod and
or xor + - (resta)
```

En un programa la claridad es fundamental, así que debemos usar paréntesis. Incluso es recomendable hacerlo en los casos en los que, por la precedencia de los operadores, no sería necesario.

Ejemplo

- $\frac{4+2}{1+1}$

lo escribimos

$$(4 + 2)/(1 + 1).$$

- Si quisiéramos escribir

$$4 + \frac{2}{1} + 1$$

bastaría  $4 + 2/1 + 1$ .

Pero es preferible ser muy claro:

$$4 + (2/1) + 1.$$

# Elementos predefinidos

Como en prácticamente cualquier lenguaje, en Pascal hay *elementos predefinidos*: funciones, operaciones y constantes definidos inicialmente en el lenguaje. No podemos declarar nuevos identificadores que usen estos nombres.

<code>abs(n)</code>	valor absoluto
<code>trunc(n)</code>	truncar a entero
<code>round(n)</code>	redondear a entero
<code>sqr(n)</code>	elevar al cuadrado
<code>sqrt(n)</code>	raíz cuadrada
<code>chr(i)</code>	carácter en posición <i>i</i>
<code>ord(c)</code>	posición del carácter o valor <i>c</i>
<code>pred(c)</code>	carácter o valor predecesor de <i>c</i>
<code>succ(c)</code>	carácter o valor sucesor de <i>c</i>
<code>arctan(n)</code>	arcotangente
<code>cos(n)</code>	coseno
<code>exp(n)</code>	exponencial
<code>ln(n)</code>	logaritmo neperiano
<code>sin(n)</code>	seno
<code>low(x)</code>	menor valor o índice en <i>x</i>
<code>high(x)</code>	mayor valor o índice en <i>x</i>

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program predefinidos;  
begin  
  writeln( abs(-3) );      // Escribe 3  
  writeln( trunc(3.64) ); // Escribe 3  
  writeln( round(3.64) ); // Escribe 4  
  writeln( sqr(3) );      // Escribe 9  
  writeln( sqrt(9) );     // Escribe 3e0  
  writeln( succ('a') );  // Escribe b  
  writeln( pred('B') );  // Escribe A  
end.
```