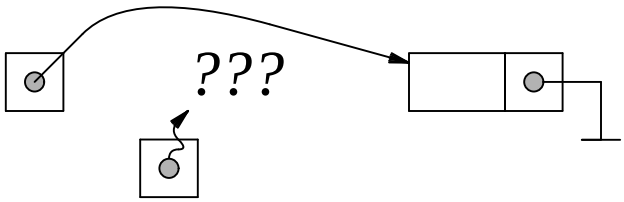
Curso Práctico de Programación



Usando Picky como Primer Lenguaje

Francisco J Ballesteros

Enrique Soriano

Gorka Guardiola

Copyright © 2011

2 — Elementos básicos

2.1. ¿Por dónde empezamos?

En el capítulo anterior hemos tomado contacto con la programación y con el lenguaje Picky. Por el momento, una buena forma de proceder es tomar un programa ya hecho (como por ejemplo el ‘‘Hola π’’ del capítulo anterior) y cambiar sólo las partes de ese programa que nos interesen. De ese modo podremos ejecutar nuestros propios programas sin necesidad de, por el momento, saber cómo escribirlos enteros por nuestros propios medios.

Recuerda que programar es como cocinar (o como conducir). Sólo se puede aprender a base de práctica y a base de ver cómo practican otros. No es posible aprenderlo realmente sólo con leer libros (aunque eso ayuda bastante). Si no tienes el compilador de Picky cerca, descárgalo de

```
http://lsub.org/ls/picky.html
```

Úsalo para probar por ti mismo cada una de las cosas que veas durante este curso. La única forma de aprender a usarlas es usándolas.

2.2. Conjuntos y elementos

El lenguaje Picky permite básicamente manipular datos. Eso sí, estos datos son entidades abstractas y están alejadas de lo que en realidad entiende el ordenador. Programar en Picky consiste en aprender a definir y manipular estas entidades abstractas, lo cual es más sencillo de lo que parece.

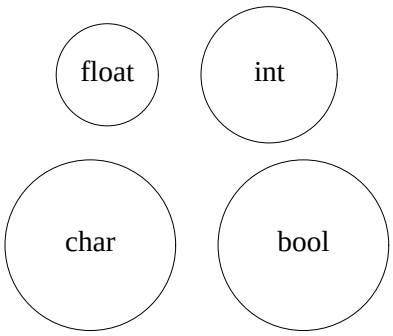


Figura 2.1: *El mundo según Picky: existen enteros, caracteres, reales y valores de verdad.*

¿Cómo es el mundo para Picky? En realidad es tan sencillo como muestra la figura 2.1. Para Picky, en el mundo existen entidades (cosas) que pueden ser números reales, números enteros, caracteres o valores de verdad. ¡Y no hay nada más! (por el momento).

Esto quiere decir que en Picky tenemos estos cuatro conjuntos diferentes (números reales, números enteros, caracteres y valores de verdad). Cada uno de ellos consta de elementos homogéneos entre sí. El conjunto de los caracteres contiene, sorprendentemente, caracteres; igual sucede con el conjunto de los números reales, que contiene números reales.

Cada conjunto tiene unas operaciones bien definidas para trabajar sobre sus elementos. Por ejemplo, podemos sumar números enteros entre sí para obtener otros números enteros. Igualmente, podemos sumar números reales entre sí para obtener otros números reales. Pero *no podemos mezclar números enteros con números reales*.

Quizá sorprenda esto, pero así son las cosas. Si queremos sumar un número entero a un número real tendremos que conseguir un número real que tenga el mismo valor que el número entero, y luego sumarlo. La razón por la que esto es así es intentar evitar que, por accidente,

hagamos cosas tales como sumar peras con manzanas; o contamos peras o contamos manzanas o contamos piezas de fruta. Pero no podemos mezclarlo todo a voluntad.

A cada conjunto se le denomina **tipo de datos** y, naturalmente, a los elementos de cada conjunto se les denomina **datos**. Resumiendo:

- Un tipo de datos es un conjunto de elementos con unas operaciones bien definidas para datos pertenecientes a ese tipo.
- No es posible combinar datos de distinto tipo entre sí.

Debido a esto se dice que Picky es un lenguaje **fuertemente tipado**. Hay otros lenguajes de programación que permiten combinar entre sí elementos de distintos tipos, pero esto da lugar a muchos errores a la hora de programar.

Picky utiliza la palabra reservada *int* para denominar al tipo de datos (conjunto) de los números enteros. El conjunto de los reales está representado en Picky por el tipo de datos *float*. De igual manera, tenemos el conjunto de los valores de verdad, ‘‘cierto’’ y ‘‘falso’’, representado en Picky por el tipo de datos *bool*. Este último sólo tiene dos elementos: *True* representa en Picky al valor de verdad ‘‘cierto’’ y *False* representa al valor de verdad ‘‘falso’’. Tenemos también un tipo de datos para el conjunto de caracteres disponible, llamado *char*.

Así, el literal 3 pertenece al tipo de datos *int* dado que es un número entero y sólo puede combinarse normalmente con otros elementos también del tipo *int*. La siguiente expresión en Picky intenta sumar dos y tres:

```
2 + 3.0                ¡incorrecto!
```

Un entero sólo puede sumarse con otro entero; nunca con un número real. El punto entre el tres y el cero del segundo operando hace que ese literal sea un número real. Por tanto, dicha expresión es incorrecta y dará lugar a un error durante la compilación del programa que la incluya. Lo hemos intentado (cambiando un poco el programa de saludo a `π`) y esto es lo que ha pasado:

```
; pick malo.p
malo.p:13: incompatible argument types (int and float) for op '+'
;
```

El compilador ha escrito un mensaje para informar de un error, intentando describirlo de la forma más precisa que ha podido, y no ha hecho nada más. Esto es, **no ha compilado el programa**. ¡No tenemos un fichero ejecutable! El programa contenía la siguiente sentencia (hemos cambiado la línea 13 de `holapi.p` para que imprima la suma del número entero con el número real):

```
writeln(2 + 3.0);
```

y esta sentencia no compila, dado que intenta mezclar elementos de distintos tipos.

La razón es que debe existir **compatibilidad de tipos** entre los elementos que aparecen en una expresión. Esto es, podemos escribir

```
2 + 3
```

y también

```
2.0 + 4.3
```

pero no expresiones que mezclen valores de tipo *int* y valores de tipo *float* como en la expresión incorrecta mostrada arriba.

Los literales de tipo *float* pueden también escribirse en notación exponencial, expresados como un número multiplicado por 10 elevado a una potencia dada. Por ejemplo,

```
3.2E-3
```

es en realidad $3.2 \bullet 10^{-3}$, esto es: 0.0032.

En aquellas ocasiones en que necesitamos sumar un entero a un número real, podemos realizar una **conversión de tipo** sobre uno de los valores para conseguir un valor equivalente en el tipo de datos que queremos utilizar. Por ejemplo,

```
float(3)
```

es una expresión cuyo valor es en realidad 3.0, de tipo *float*. Las conversiones de tipo tienen siempre este aspecto, pero no siempre se permiten. Por ejemplo, no podemos convertir un carácter a un número real, igual que no tendría sentido convertir una pera en una manzana o el plomo en oro (bueno, esto último tal vez sí).

2.3. Operaciones

Como estamos viendo, cada tipo de datos tiene un nombre, por ejemplo *int*, y define un conjunto de elementos de dicho tipo. Además, cada tipo de datos tiene unas operaciones básicas que nos permiten manipular datos de dicho tipo. El reflejo en Picky de estas operaciones básicas (que son en realidad operaciones matemáticas) son los llamados **operadores**. Por ejemplo, ‘‘`+`’’ es un operador para el tipo *int* y podemos utilizarlo para sumar números enteros.

Un operador permite operar sobre uno o más elementos de un tipo de datos determinado. A estos elementos se les denomina **operandos**. Algunos operadores se escriben entre los operandos (por ejemplo, la suma) y se les denomina **infijos**. Otros operadores adoptan otras formas. Por ejemplo, el cambio de signo se escribe con un ‘‘`-`’’ escrito antes del objeto sobre el que opera; por eso se denomina operador **prefijo**. Otros operadores adoptan el aspecto de funciones, como por ejemplo el operador de conversión de tipo que hemos visto antes para convertir un 3 en un valor de tipo *float*.

Los tipos de datos numéricos admiten los siguientes operadores:

- Suma: `3 + 4`
- Resta: `4.0 - 7.2`
- Multiplicación: `2 * 2`
- División: `7 / 3`
- Exponencial: `2 ** 3` (Esto es, 2^3)
- Conversión a otro tipo numérico: `float(3)`, `int(3.5)`
- Cambio de signo: `- 5`

En realidad, tenemos un juego de estos operadores para los enteros y otro distinto para los reales, aunque se llamen igual. Así, la división es concretamente una división entera (parte entera del cociente) cuando los números son enteros y es una división real en otro caso.

El tipo *int* dispone además del siguiente operador:

- Módulo: `5 % 3`

El módulo corresponde con el resto de la división entera. Por tanto, sólo está disponible para el tipo de datos *int*.

La lista de operadores la hemos incluido aquí más como una referencia que como otra cosa. Podemos olvidarnos de ella por ahora. A medida que los uses los recordarás de forma natural.

2.4. Expresiones

Empleando los operadores que hemos descrito es posible escribir expresiones más complejas que calculen valores de forma similar a como hacemos en matemáticas. El problema de la escritura de expresiones radica en que para Picky el programa es simplemente una única secuencia de caracteres. Así es como ve Picky el programa para saludar la número π:

```
/*\n→enSaludar al numero π.\n */\n\nprogram holapi;\n\nconsts:
\n→enPi = 3.1415926;\n\nprocedure main()\n{\n→enwrite("Hola ");
\n→enwrite(Pi);\n→enwrite("!");\n→enwriteeol();\n}\n
```

Lo ve como si todas las líneas del fichero estuvieran escritas en una única línea extremadamente larga. Esto quiere decir que no podemos escribir un 3 sobre el signo de dividir y una suma bajo el mismo para expresar que el denominador es una suma. Por ejemplo, para dividir 3 por 2 tenemos que escribir

```
3 / 2
```

pero no podemos escribir $\frac{3}{2}$.

Igualmente, para sumar los números del 1 al 5 y dividir el resultado por 3 no podemos utilizar

$$\frac{1+2+3+4+5}{3}$$

En su lugar, es preciso escribir:

```
(1 + 2 + 3 + 4 + 5) / 3
```

Los paréntesis son necesarios para agrupar la expresión que suma los números del uno al cinco, de tal forma todas las sumas estén en el numerador.

Lo que pasa es que los operadores se evalúan empleando un orden determinado. Por eso se dice que algunos tienen **precedencia** sobre otros (que el lenguaje los escoge y los evalúa antes que estos otros). Por ejemplo, la división y la multiplicación tienen precedencia sobre las sumas y restas. Esto es, las divisiones y las multiplicaciones en una expresión se calculan antes que las sumas y las restas. En el ejemplo anterior, de no utilizar los paréntesis estaríamos calculando en realidad:

```
1 + 2 + 3 + 4 + 5 / 3
```

Esto es,

```
1 + 2 + 3 + 4 + (5 / 3)
```

O lo que es lo mismo:

$$1+2+3+4+\frac{5}{3}$$

La exponenciación tiene precedencia sobre la multiplicación y la división. Por tanto,

```
2 * 3 ** 2
```

está calculando

```
2 * (3 ** 2)
```

y no

```
(2 * 3) ** 2
```

Cuando existan dudas sobre la precedencia conviene utilizar paréntesis para agrupar las expresiones según deseemos.

Una nota importante es que *se debe utilizar el espacio en blanco y los signos de puntuación (paréntesis en este caso) para hacer más legible la expresión.* Como puede verse en las expresiones de ejemplo, los espacios están escritos de tal modo que resulta más fácil leerlas. Normalmente, se escriben antes y después del operador, pero nunca después de un paréntesis abierto o antes de uno cerrado.

2.5. Otros tipos de datos

El tipo de datos *char* representa un carácter del juego de caracteres disponibles. Estos son algunos ejemplos de literales de tipo carácter:

```
'A'
'0'
' '
```

El primero representa a la letra ‘‘A’’, el segundo al dígito ‘‘0’’ y el último al espacio en blanco. Todos ellos son de tipo *char*. Es importante ver que `'0'` es un carácter y `0` es un entero. El primero se usa al manipular texto y el segundo al manipular números. ¡No tienen que ver entre si! De hecho... ¡No pueden operarse entre sí dado que son de distinto tipo!

Un carácter se almacena en el ordenador como un número cuyo valor representa el carácter en cuestión. Inicialmente se utilizaba el código **ASCII** para representar los caracteres. Hoy día son populares otros códigos como **UTF**. Una codificación de caracteres no es más que una tabla en la que se asigna un valor numérico a cada carácter que se quiera representar (como ya sabemos, los ordenadores sólo saben manejar números). En la mayoría de los casos, las letras sin acentuar, dígitos y signos comunes de puntuación empleados en Inglés están disponibles en el juego de caracteres ASCII. Suele ser buena idea *no* emplear acentos en los identificadores que empleamos en los programas por esta razón. Aunque Picky lo permite, no es así en otros lenguajes y no todo el software de ordenador (editores por ejemplo) se comporta igual con caracteres acentuados.

En ocasiones, es útil saber qué posición ocupa un carácter en la tabla ASCII. En Picky podemos obtener esta correspondencia mediante una conversión explícita del carácter a un entero. Por ejemplo, para saber la posición en la tabla del carácter `'Z'`, podemos usar la expresión:

```
int('Z')
```

Dicha expresión tiene como valor `90` dado que `'Z'` ocupa esa posición en el código. Para conseguir lo inverso, esto es, conseguir el carácter que corresponde a una posición en la tabla, podemos utilizar la conversión a *char*. Por ejemplo, esta expresión

```
char(90)
```

tiene como valor `'Z'`, del tipo *char*, dado que ese carácter tiene la posición 90 en el código de caracteres.

Otro tipo de datos importante es el llamado booleano, *bool* en Picky. Este tipo representa valores de verdad: ‘‘Cierto’’ y ‘‘Falso’’. Está compuesto sólo por el conjunto de elementos *True* y *False*, que corresponden a ‘‘Cierto’’ y ‘‘Falso’’.

¿Recuerdas la selección en los algoritmos? Este tipo es realmente útil dado que se emplea para expresar condiciones que pueden cumplirse o no, y para hacer que los programas ejecuten unas sentencias u otras dependiendo de una condición dada.

Los operadores disponibles para este tipo son los existentes en el llamado **álgebra de Boole** (de ahí el nombre del tipo).

- Negación: `not`
- Conjunción: `and`
- Disyunción: `or`

Sus nombres son pintorescos pero es muy sencillo hacerse con ellos. Sabemos que *True* representa algo que es cierto y *False* representa algo que es falso. Luego si algo no es falso, es que es cierto. Y si algo no es cierto, es que es falso. Esto es:

```
not True == False
not False == True
```

Sigamos con la conjunción. Si algo que nos dicen es en parte verdad y en parte mentira... ¿Nos

están mintiendo? Si queremos ver si dos cosas son conjuntamente verdad utilizamos la conjunción. Sólo es cierta una conjunción de cosas ciertas:

```
False and False == False
False and True == False
True and False == False
True and True == True
```

Sólo resulta ser verdad *True and True*. ¡El resto de combinaciones son falsas!

Ahora a por la disyunción. Esta nos dice que alguna de dos cosas es cierta (o tal vez las dos). Esto es:

```
False or False == False
False or True == True
True or False == True
True or True == True
```

Con algún ejemplo más todo esto resultará trivial. Sea *pollofrito* un valor que representa que un pollo está frito y *pollocrudo* un valor que representa que un pollo está crudo. Esto es, si el pollo está frito tendremos que

```
pollofrito == True
pollocrudo == False
```

Eso sí, cuando el pollo esté crudo lo que ocurrirá será mas bien

```
pollofrito == False
pollocrudo == True
```

Luego ahora odemos decir que

```
pollofrito or pollocrudo
```

es siempre *True*. Pero eso sí, ¡bajo ningún concepto!, ni por encima del cadáver del pollo, podemos conseguir que

```
pollofrito and pollocrudo
```

sea cierto. Esto va a ser siempre *False*. O está frito o está crudo. Pero hemos quedado que en este mundo binario nunca vamos a tener ambas cosas a la vez.

Si sólo nos comemos un pollo cuando está frito y el valor *polloingerido* representa que nos hemos comido el pollo, podríamos ver que

```
pollofrito and polloingerido
```

bien podría ser cierto. Pero desde luego

```
pollocrudo and polloingerido
```

tiene mas bien aspecto de ser falso.

Los booleanos son extremadamente importantes como estamos empezando a ver. Son lo que nos permite que un programa tome decisiones en función de cómo están las cosas. Por lo tanto tenemos que dominarlos realmente bien para hacer buenos programas.

Tenemos también una gama de operadores que ya conoces de matemáticas que aquí producen como resultado un valor de verdad. Estos son los operadores de comparación, llamados así puesto que se utilizan para comparar valores entre sí:

```
<
>
<=
>=
==
!=
```

Podemos utilizarlos para comparar valores numéricos, caracteres y booleanos. Pero ambos operandos han de ser del mismo tipo. Estos operadores corresponden a las relaciones *menor que*, *mayor que*, *menor o igual que*, *mayor o igual que*, *igual a* y *distinto a*. `3 < 2` es *False*, pero `2 <= 2` es *True*.

Por ejemplo, sabemos que si *a* y *b* son dos *int*, entonces

```
(a < b) or (a == b) or (a > b)
```

va a ser siempre *True*. No hay ninguna otra posibilidad.

Cuando queramos que un programa tome una decisión adecuada al considerar alguna condición nos vamos a tener que plantear *todos los casos posibles*. Y vamos a tener que aprender a escribir expresiones booleanas para los casos que nos interesen.

Pero cuidado, los números reales se almacenan como aproximaciones. Piénsese por ejemplo en como si no podríamos almacenar el número π, que tiene infinitas cifras decimales. Por ello no es recomendable comparar números reales empleando ‘‘`==`’’ o ‘‘`!=`’’. Ya veremos más adelante cómo podemos comparar números reales.

La comparación de igualdad y desigualdad suele estar disponible en general para todos los tipos de datos. Las comparaciones que requieren un orden entre los elementos comparados suelen estar disponibles en tipos de datos numéricos.

Quizá sorprenda que también pueden compararse caracteres. Por ejemplo, esta expresión es cierta, *True*, cuando *x* es un carácter que corresponde a una letra mayúscula:

```
('A' <= x) and (x <= 'Z')
```

Ya comentamos antes que los caracteres son en realidad posiciones en una tabla. Las letras mayúscula están en dicha tabla de caracteres en posiciones consecutivas. Igual sucede con las minúsculas. Los dígitos también. Por tanto, podemos compararlos en base a su posición en la tabla. Pero cuidado, esta expresión no tiene mucho sentido:

```
x <= '0'
```

No sabemos qué caracteres hay antes del carácter `'0'` en el código ASCII (salvo si miramos la tabla que describe dicho código).

Otro ejemplo más. Debido al tipado estricto de datos en Picky no podemos comparar

```
3 < '0'
```

dado que no hay **concordancia de tipos** entre `3` y `'0'`. El primero es de tipo entero, y el segundo de tipo carácter. Esto es, dado que los tipos no son compatibles para usarlos en el mismo operador ‘‘`<`’’.

2.6. Años bisiestos

Podemos combinar todos estos operadores en expresiones más complicadas. Y así lo haremos con frecuencia. Por ejemplo, sea *a* un entero que representa un año, como 2008 o cualquier otro. Esta expresión es *True* si el año *a* es bisiesto:

```
(a % 4) == 0 and ((a % 100) != 0 or (a % 400) == 0)
```

¿De dónde hemos sacado esto? Los años múltiplos de 4 son bisiestos, excepto los que son

múltiplos de 100. Salvo por que los múltiplos de 400 lo son. Si queremos una expresión que sea cierta cuando el año es múltiplo de 4 podemos fijarnos en el módulo entre 4. Este será 0, 1, 2 o 3. Para los múltiplos de 4 va a ser siempre 0. Luego

```
(a % 4) == 0
```

es *True* si *a* es múltiplo de 4. Queremos excluir de nuestra expresión aquellos que son múltiplos de 100, puesto que no son bisiestos. Podemos escribir entonces una expresión que diga que *el módulo entre 4 es cero y no es cero el módulo entre 100*. Esto es:

```
(a % 4) == 0 and not ((a % 100) == 0)
```

Esta expresión es cierta para todos los múltiplos de 4 salvo que sean también múltiplos de 100. Pero esto es igual que

```
(a % 4) == 0 and (a % 100) != 0
```

que es más sencilla y se entiende mejor. Pero tenemos que hacer que para los múltiplos de 400, a pesar de ser múltiplos de 100, la expresión sea cierta. Dichos años son bisiestos y tenemos que considerarlos. Veamos: suponiendo que *a* es múltiplo de 4, lo que es un candidato a año bisiesto... Si *a* es no múltiplo de 100 o *a* es múltiplo de 400 entonces tenemos un año bisiesto. Esto lo podemos escribir como

```
(a % 100) != 0 or (a % 400) != 0
```

pero tenemos naturalmente que exigir que nuestra suposición inicial (que *a* es múltiplo de 4) sea cierta:

```
(a % 4) == 0 and ((a % 100) != 0 or (a % 400) == 0)
```

¿Cómo hemos procedido para ver si *a* es bisiesto? Lo primero ha sido definir el problema: Queremos *True* cuando *a* es bisiesto y *False* cuando no lo es. Ese es nuestro problema.

Una vez hecho esto, hemos tenido que saber cómo lo haríamos nosotros. En este caso lo mas natural es que no sepamos hacerlo (puesto que normalmente miramos directamente el calendario y no nos preocupamos de calcular estas cosas). Así pues tenemos que aprender a hacerlo nosotros antes siquiera de pensar en programarlo. Tras buscar un poco, aprendemos mirando en algún sitio (¿Google?) que

> *''Los años múltiplos de 4 son bisiestos, excepto los que son múltiplos de 100. Salvo por que los múltiplos de 400 lo son.''*

Nuestro plan es escribir esta definición en Picky y ver cuál es su valor (cierto o falso). Ahora tenemos que escribir el programa. Lo más importante es en realidad escribir la expresión en Picky que implementa nuestro plan. Procediendo como hemos visto antes, llegamos a nuestra expresión

```
(a % 4) == 0 and ((a % 100) != 0 or (a % 400) == 0)
```

Por cierto, hemos usado paréntesis por claridad, pero podríamos haber escrito:

```
a%4 == 0 and (a%100 != 0 or a%400 == 0)
```

¡Ahora hay que probarlo! ¿Será e 2527 un año bisiesto? ¿Y el año 1942? Tomamos prestado el programa para saludar a π y lo cambiamos para que en lugar de saludar a π escriba esta expresión. Por el momento no sabemos lo necesario para hacer esto (dado que no hemos visto ningún programa que escriba valores de verdad). En cualquier caso, este sería el programa resultante.

esbisiesto.p

```
1    /*
2     *   Es 2527 un año bisiesto?
3     */
```

```
5     program esbisiesto;

7     consts:
8           /*
9            *   A es el año que nos interesa
10           */
11          A = 2527;

13          /*
14           *   Esta constante booleana será cierta si A es bisiesto
15           */
16          EsABisiesto = (A % 4) == 0 and ((A % 100) != 0 or (A % 400) == 0);
17
18    /*
19     *   Programa Principal.
20     */

22    procedure main()
23    {
24          writeln(EsABisiesto);
25    }
—
```

Lo que resta es compilar y ejecutar el programa:

```
; pick esbisiesto.p
; out.pam
False
```

Cuando se ha compilado el programa, el complilador ha calculado el valor de las expresiones cuyos operandos son constantes (todas las de este programa tienen operandos constantes), y ha generado el fichero binario con las instrucciones del programa y los datos que necesita (incluyendo las constantes que hemos definido).

Cuando se ha ejecutado el binario, se ha empezado ejecutando su procedimiento principal. En este caso, el programa principal tiene una única sentencia. Dicha sentencia ha escrito el valor de nuestra constante *EsABisiesto* como resultado (salida) del programa.

Modifica tú este programa para ver si 1942 es bisiesto o no. Intenta cambiarlo para que un sólo programa te diga si dos años que te gusten son bisiestos.

2.7. Más sobre expresiones

¿Cómo se calcula en Picky el valor de las constantes? Se calcula el valor correspondiente a la expresión durante la compilación del programa. Calcular el valor de una expresión se denomina **evaluar** la expresión. La constante queda definida con el valor resultante tras evaluar la expresión. En el últimpo ejemplo, el compilador ha calculado un valor de tipo *bool* para la expresión a partir de la cual hemos definido la constante *EsABisiesto*. Luego la constante *EsABisiesto* será de tipo *bool*. Cuando el programa ejecute, siempre que use la constante definida, esta tendrá el mismo valor (el que se calculó cuando se compiló el programa).

Nótese que no todas las expresiones que aparecen en un programa se calculan al compilar el programa. Si la expresión tiene operandos que no son constantes, la expresión se evalúa al ejecutar el programa. En realidad, ese será el caso común, como veremos más adelante en el curso. En todo caso, la evaluación de una expresión se realiza de la misma forma en ambos casos.

Las expresiones se evalúan siempre *de dentro hacia afuera*, aunque no sabemos si se evalúan de derecha a izquierda o de izquierda a derecha. Esto quiere decir que las expresiones se evalúan haciendo caso de los paréntesis (y de la precedencia de los operadores) de tal forma que primero se hacen los cálculos más interiores o **anidados** en la expresión. Por ejemplo, si partimos

de

```
(A % 4) == 0 and ((A % 100) != 0 or (A % 400) == 0)
```

y *A* tiene como valor `444`, entonces Picky evalúa

```
(444 % 4) = 0 and ((444 % 100) != 0 or ((444 % 400) == 0))
```

Aquí, Picky va a calcular primero `444 % 4` o bien `444 % 100` o bien `444 % 400`. Estas subexpresiones son las más internas o más anidadas de la expresión. Supongamos que tomamos la segunda. Esto nos deja:

```
(444 % 4) == 0 and (44 != 0 or ((444 % 400) == 0))
```

Picky seguiría eliminando paréntesis (evaluándolos) de dentro hacia afuera del mismo modo, calculando...

```
(444 % 4) == 0 and (44 != 0 or (44 == 0))
(444 % 4) == 0 and (44 != 0 or False)
(444 % 4) == 0 and (True or False)
0 == 0 and (True or False)
True and (True or False)
True and True
True
```

Si no tenemos paréntesis recurrimos a la precedencia para ver en qué orden se evalúan las cosas. En la figura 2.2 mostramos todos los operadores, de mayor a menor precedencia (los que están en la misma fila tienen la misma precedencia). En Picky, los operadores con la misma precedencia se evalúan de izquierda a derecha (ten cuidado, esto no es así en todos los lenguajes de programación).

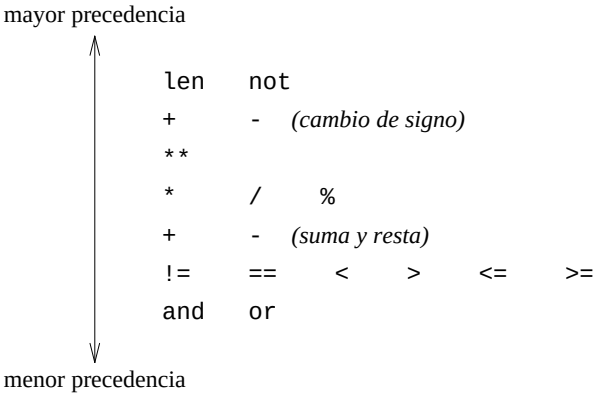


Figura 2.2: *Precedencia de operadores en Picky.*

Si tomamos por ejemplo la expresión

```
not False and (False or True)
```

podemos ver que *not* va antes que *and*, por lo que en realidad tenemos

```
((not False) and (False or True))
```

Para evaluarla nos fijamos en partes de la expresión de dentro de los paréntesis hacia afuera y vamos reescribiendo la expresión de tal forma que cambiamos cada parte por su valor. Por ejemplo:

```
((not False) and (False or True))
(True and (False or True))
(True and True)
True
```

Por cierto, hay dos leyes muy útiles en expresiones booleanas que son las denominadas **leyes de De Morgan** y que podemos utilizar para simplificar expresiones booleanas. Según estas leyes, resulta que las siguientes expresiones son equivalentes:

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

Y las siguientes son también equivalentes:

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

Puedes comprobarlo. Sugerimos intentar eliminar los *not* de las condiciones en los programas, puesto que los humanos entienden peor las negaciones que las afirmaciones ¿No te parece que no sucede que no es cierto? O tal vez deberíamos decir... ¿No es cierto?

Un último detalle importante sobre las expresiones. Como ya se explicó en el capítulo anterior, el compilador hace todo lo posible por detectar fallos en el programa. Algunos errores en expresiones se detectan al compilar el programa, generando un error de compilación.

El compilador evalúa las expresiones cuyos operandos son constantes, ya que los valores de las constantes se conocen cuando el compilador traduce el programa. Gracias a esto, el compilador es capaz de detectar ciertos errores en las expresiones evitando que el programa tenga errores de ejecución. Por ejemplo, si en una expresión se está dividiendo entre una constante con valor cero o se está calculando la raíz cuadrada de una constante con un valor negativo, el compilador se negará a generar el fichero binario y dará errores de compilación. Pero atención, esto no nos libra de tener otros errores lógicos o de ejecución en nuestros programas.

2.8. Elementos predefinidos en Picky

En Picky existen funciones, operaciones y constantes predefinidas en el lenguaje (en inglés, *built-in*) que resultan muy útiles para construir expresiones. Estas funciones y constantes pueden ayudarnos a obtener valores que dependen de un tipo de datos determinado.

Los tipos ordinales, esto es, los tipos de datos cuyos valores se pueden contar y tienen un orden, tienen definidas las funciones

```
pred(v)
succ(v)
```

que, respectivamente, evalúan al valor predecesor (anterior) y sucesor (posterior) al valor *v* (en el tipo de datos correspondiente). Por ejemplo, si *c* es de tipo *char* con valor `'B'`, entonces

```
pred(c)
```

tiene como valor `'A'`. Siguiendo con el mismo ejemplo,

```
succ(c)
```

tiene como valor el siguiente valor en el tipo de datos *char*, que es `'C'`.

También tenemos funciones predefinidas para los números reales que son muy útiles para escribir expresiones, como la raíz cuadrada, el seno, el coseno, etc.

Built-in	Función
`acos(r)`	arcocoseno
`asin(r)`	arcoseno
`atan(r)`	arcotangente
`cos(r)`	coseno
`exp(r)`	exponencial
`log(r)`	logaritmo
`log10(r)`	logaritmo base 10
`pow(r1, r2)`	potencia
`sin(r)`	seno
`sqrt(r)`	raíz cuadrada
`tan(r)`	tangente

Figura 2.3: *Funciones predefinidas para números reales.*

Un *built-in* llamado *fatal* nos permite abortar la ejecución del programa en cualquier punto. Esto sólo se debe hacer cuando nuestro programa ha detectado una situación de la que no se puede recuperar. Ya veremos ejemplos durante el curso. Por ejemplo, si el programa llega a ejecutar la sentencia

```
fatal("Error muy grave!");
```

entonces se acaba su ejecución y se imprime por la pantalla ‘‘`fatal: Error muy grave!`’’. A medida que avance el libro, irán apareciendo nuevos *built-in*.

Picky también tiene constantes predefinidas. Por ejemplo, Picky tiene constantes para el mínimo y el máximo valor de los tipos *int* y *char*. Así podemos utilizar estos valores en las expresiones que construimos. *Minint* representa el mínimo valor entero y *Maxint* representa el máximo valor entero. Por ejemplo, estas dos sentencias escribirían por la pantalla el máximo entero y su predecesor:

```
writeln(Maxint);
writeln(pred(Maxint));
```

Igualmente, *Minchar* y *Maxchar* representan el mínimo y el máximo valor de un carácter.

No podemos usar los nombres de las funciones y las constantes predefinidas como identificadores en nuestros programas, ya que esos nombres están reservados para el propio lenguaje (como las palabras reservadas).

Los *built-ins* se pueden usar en cualquier expresión del programa, incluyendo las expresiones que escribimos para definir nuestras propias constantes. Por ejemplo:

```
consts:
      C = sqrt(120.0) + 22.0;
```

En este ejemplo, la constante real *C* tendrá el valor resultante de sumar 22.0 a la raíz cuadrada de 120.0.

2.9. Longitud de una circunferencia

Vamos a poner todo esto en práctica. Queremos un programa que escriba la longitud de una circunferencia de radio *r*. El problema es: dado *r*, calcular $2\pi r$, naturalmente. Podríamos modificar el último programa que hemos hecho para conseguir esto. Pero vamos a escribirlo desde el principio. Lo primero que necesitamos es escribir la estructura de nuestro programa. Podemos empezar por...

calculocircunferencia.p

```
1     /*
2      *    Programa para calcular la longitud de
3      *    la circunferencia de un circulo de radio r.
4      */

6     program calculocircunferencia;

8     /*
9      *    Programa Principal.
10     */
11    procedure main()
12    {
13         ;
14    }
```

No es que haga mucho. De hecho, hemos utilizado la sentencia nula como cuerpo del programa principal: dejando únicamente el punto y coma. Esto quiere decir que Picky no va a ejecutar nada cuando ejecute el cuerpo del programa. Ahora compilamos y ejecutamos el programa para ver al menos que está bien escrito. Evidentemente, no veremos nada en la pantalla al ejecutar el programa, porque no hace nada.

Lo siguiente que necesitamos es nuestra constante π. Podemos utilizar la que teníamos escrita en otros programas y ponerla al principio en la zona de declaraciones de constantes.

calculocircunferencia.p

```
1     /*
2      *    Programa para calcular la longitud de
3      *    la circunferencia de un circulo de radio r.
4      */

6     program calculocircunferencia;

8     consts:
9          /*
10          *    Número Pi
11          */
12         Pi = 3.1415926;

14    /*
15     *    Programa Principal.
16     */
17    procedure main()
18    {
19         ;
20    }
```

De nuevo compilamos y ejecutamos el programa. Al menos sabremos si tiene errores sintácticos o no. Si los tiene será mucho más fácil encontrar los errores ahora que luego.

Ahora podríamos definir otra constante para el radio que nos interesa para el cálculo, *Radio*, y otra más para la expresión que queremos calcular. Podríamos incluir esto justo debajo de la declaración para *Pi*:

```
16     Radio = 3.0;
17     LongitudCircunferencia = 2.0 * Pi * Radio;
```

Por último necesitamos escribir el resultado de nuestro cálculo. Para ello cambiamos la sentencia

nula por

```
writeln(LongitudCircunferencia);
```

en el cuerpo del programa. El programa resultante es como sigue:

calculocircunferencia.p

```
1   /*
2    *   Programa para calcular la longitud de
3    *   la circunferencia de un circulo de radio r.
4    */

6   program calculocircunferencia;

8   consts:
9        /*
10        *   Número Pi
11        */
12       Pi = 3.1415926;
13       /*
14        *   Radio que nos interesa y long. circunf.
15        */
16       Radio = 3.0;
17       LongitudCircunferencia = 2.0 * Pi * Radio;

19  /*
20   *   Programa Principal.
21   */

23  procedure main()
24  {
25       writeln(LongitudCircunferencia);
26  }
```

Si lo compilamos de nuevo y lo ejecutamos tenemos el resultado.

```
; pick calculocircunferencia.p
; out.pam
18.849556
```

Este programa está bien. Pero lo suyo es que ya que la longitud de una circunferencia es en realidad una función, el programa utilice una función para calcularla (como el programa que utilizamos para ver la anatomía de un programa al principio del curso). Podemos copiar la función que aparecía en ese programa y adaptarla a nuestros propósitos. Ésta nos quedaría así:

```
18  /*
19   *   Función que calcula la longitud de la
20   *   circunferencia de radio r.
21   */
22  function longitudcircunferencia(r: float): float
23  {
24       return 2.0 * Pi * r;
25  }
```

La sintaxis puede ser rara, pero es fácil de entender. Esto define una función llamada *longitudcircunferencia* a la que podemos dar un número de tipo *float* para que nos devuelva otro número de tipo *float*. El número que le damos a la función lo llamamos *r*. Entre ‘‘{’’ y ‘‘}’’ tenemos que escribir cómo se calcula el valor de la función. La sentencia *return* hace que la función devuelva el valor escrito a su derecha ($2\pi r$) cada vez que le demos un número *r*. Si

utilizamos esto, el programa quedaría como sigue:

calculocircunferencia.p

```
1     /*
2      *    Programa para calcular la longitud de
3      *    la circunferencia de un circulo de radio r.
4      */

6     program calculocircunferencia;

8     consts:
9          /*
10          *    Número Pi.
11          */
12         Pi = 3.1415926;

14         /*
15          *    Radio que nos interesa.
16          */
17         Radio = 3.0;

19    /*
20     *    Función que calcula la longitud de la
21     *    circunferencia de radio r.
22     */
23    function longitudcircunferencia(r: float): float
24    {
25         return 2.0 * Pi * r;
26    }

28    /*
29     *    Programa principal.
30     */
31    procedure main()
32    {
33         write("Longitud circunferencia= ");
34         write(longitudcircunferencia(Radio));
35         writeeol();
36    }
```

En el cuerpo del programa escribimos un mensaje explicativo, después escribimos el valor de la expresión para calcular la longitud de la circunferencia y, por último, pasamos a una línea nueva en la pantalla (escribimos un fin de línea). Para escribir por pantalla la longitud se ha escrito una expresión que usa sólo la función que acabamos de definir:

```
write(longitudcircunferencia(Radio));
```

La expresión

```
longitudcircunferencia(Radio)
```

tiene como valor (calculará) la longitud de la circunferencia de radio *Radio* (*Radio* es la constante que tenemos definida con el radio que nos interesa, en este caso 3.0). Esta es la compilación y ejecución del programa:

```
; pick calculocircunferencia.p
; out.pam
Longitud circunferencia= 18.849556
```

En adelante utilizaremos funciones para realizar nuestros cálculos, como en este último programa.

Problemas

1 Modifica el último programa que hemos mostrado para que calcule el área de un circulo de radio dado en lugar de la longitud de la circunferencia dado el radio. *Presta atención a las normas de estilo.* Esto quiere decir que tendrás que cambiar el nombre del programa y tal vez otras cosas además de simplemente cambiar el cálculo.

2 Escribe en Picky las siguientes expresiones. Recuerda que para que todos los números utilizados sean números reales has de poner siempre su parte decimal, aunque ésta sea cero. En algunos casos te hará falta utilizar funciones numéricas predefinidas en Picky, tales como la raíz cuadrada. Consulta el libro para encontrar las funciones necesarias.

a)

$$2.7^2 + -3.2^2$$

b) Para al menos 3 valores de *r*,

$$\frac{4}{3}\pi r^3$$

c) El factorial de 5, que suele escribirse como $5!$. El factorial de un número natural es dicho número multiplicado por todos los naturales menores que el hasta el 1. (Por definición, $0!$ se supone que tiene como valor 1).

d) Dados $a = 4$ y $b = 3$

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{a!}{b!(a-b)!}$$

e)

$$\sqrt{\pi}$$

f) Para varios valores de x entre 0 y 2π,

$$sen^2 x + \cos^2 x$$

g) Siendo $x = 2$,

$$\frac{1}{\sqrt{3.5x^2 + 4.7x + 9.3}}$$

3 Por cada una de las expresiones anteriores escribe un programa en Picky que imprima su valor.

a) Hazlo primero declarando una constante de prueba que tenga como valor el de la expresión a calcular.

b) Hazlo ahora utilizando una función para calcular la expresión. En los casos en que necesites funciones de más de un argumento puedes utilizar el ejemplo que sigue:

```
1 function sumar(a: float, b:float): float
2 {
3     return a + b;
4 }
```

4 Para cada una de las expresiones anteriores indica cómo se evalúan éstas, escribiendo cómo queda la expresión tras cada paso elemental de evaluación hasta la obtención de un único valor como resultado final.