Trabajando con datos. Tipos básicos

PID_00268296

Autores que han participado colectivamente en esta obra
Lluís Beltrà
Kenneth Capseta
Maria Jesús Marco Galindo
Antonio Ponce

Idea y dirección de la obra
Maria Jesús Marco Galindo

Diseño y edición gráfica
Asunción Muñoz

Material docente de la UOC

Universitat Oberta de Catalunya

Tercera edición: febrero 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Contenidos

Trabajando con datos

Tipos básicos de datos

Aunque aún no sabemos cómo diseñar algoritmos, lo que sí sabemos es que deberán tratar con datos. De hecho, tienen que procesar y gestionar toda la información necesaria para resolver el problema planteado. Por lo tanto, tenemos que saber representar y tratar los datos de partida del problema (datos de entrada) y los que pide la solución del problema (datos de salida). También necesitamos saber cómo podemos leer los datos de entrada y cómo podemos devolver los datos de salida.

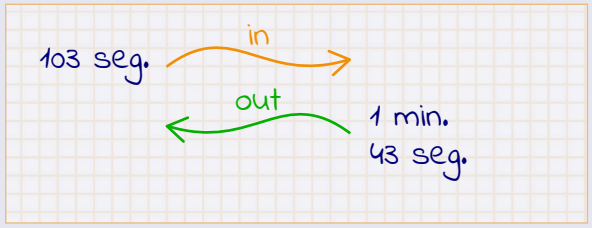
A estas alturas ya hemos entendido el problema a resolver (paso 1), por lo que ya tenemos claro qué tiene que hacer el algoritmo, qué datos de entrada tendrá y qué datos de salida deberá generar para resolver lo que nos piden. También hemos pensado cómo resolverlo, así que tenemos decidida una estrategia de pasos para solucionar el problema (paso 2). Ya podemos **diseñar el algoritmo** (paso 3), es decir, explicar la solución que hemos pensado en un algoritmo utilizando la notación algorítmica básica. Recordad que también lo podríamos hacer con un diagrama de flujo.

Ejemplo

Imaginemos que nos llama un amigo al que hace tiempo que no vemos. Nos pasamos mucho rato al teléfono, nos hemos puesto al día. Justo antes de colgar leemos en la pantalla "**103 SEC**". Qué extraño, ¿nos indica el tiempo de la llamada en segundos? Qué móvil más curioso. ¿Y esto cuántos minutos son? Haremos un algoritmo que convierta los segundos en minutos y seguro que así lo entenderemos mejor.

1 Entendemos el problema

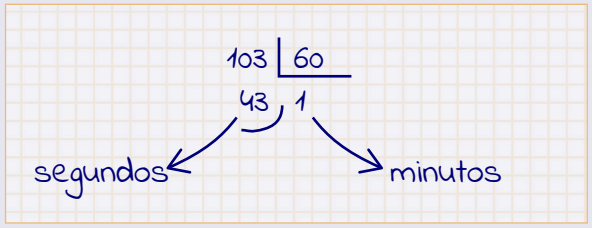
De entrada, tendremos un número que serán los segundos de la llamada y queremos calcular y dar como resultado otros dos números que sean este mismo tiempo, pero expresado en minutos y segundos:



Ya tenemos claro el problema que resolver.

2 Pensamos cómo resolverlo

Para calcular los minutos y segundos que hay en 103 segundos, debemos agrupar los segundos de 60 en 60 y así sabremos cuántos minutos exactos son, y el resto serán los segundos que sobran. Esto lo calculamos con una división:



Ya tenemos el modo (la estrategia) para hallar la solución a nuestro problema.

3 Diseñamos el algoritmo

Ahora tenemos que plasmar esta estrategia en un algoritmo. Diseñar (escribir) un algoritmo que indique claramente los pasos a seguir para resolver el problema, utilizando los elementos algorítmicos básicos tanto para representar los datos que sean necesarios como para indicar las instrucciones a seguir en cada paso.

Primera pregunta a resolver: ¿cómo representamos los minutos? ¿Y los segundos? Necesitamos saber cómo representar los datos algorítmicamente.

Variables

Cuando diseñamos el algoritmo, necesitamos referenciar los datos que trata de algún modo, representarlos algorítmicamente.

Concepto clave

Recordad que la algorítmica maneja tres elementos de tratamiento de datos:
ENTRADA: indica al usuario que se puede introducir un dato, normalmente desde el teclado.
SALIDA: indica que se mostrará un dato, normalmente en pantalla.
GUARDAR/RECORDAR: registrar un dato para que la pueda tratar el algoritmo.

Un algoritmo necesita recordar la información de entrada, como el número 103 del ejemplo, para poderlo tratar (para utilizarlo en la división, en este caso) y también el que va obteniendo para solucionar el problema, los datos intermedios (el cociente y el resto de la división, si seguimos con el ejemplo). Las variables se utilizan para almacenar esta información. Una **variable** se puede ver como una caja donde se deja cada uno de los datos que va usando el algoritmo.

Para que un algoritmo pueda trabajar con una variable necesita tener un **nombre**, para poder referirse a ella, saber qué **tipo** de datos puede contener y el **valor** del dato. Una variable siempre está definida por estos tres elementos. El valor guardado en una variable puede cambiar ("variar"), porque el algoritmo lo modifique con los tratamientos que realice.

A veces, sin embargo, el algoritmo necesita trabajar con valores constantes, que sabemos con certeza que no van a cambiar con las operaciones que haga el algoritmo, entonces usaremos **constantes** para almacenar este valor. Una constante es igualmente una caja en la que guardar un dato de un tipo y que se identifica con un nombre, pero esta caja "guarda" un **valor predeterminado y fijo** durante todo el algoritmo, de ahí su nombre.

Sintaxis para declarar variables y constantes

```
var
    nombre: tipo;
fvar
const
    nombre: tipo := valor;
fconst
```

Las palabras en negrita son palabras reservadas del lenguaje algorítmico, esto significa que son palabras clave que una máquina conoce y que podrá interpretar.

Hay diferentes tipos de datos. Los básicos son: entero, real, palabra, carácter.

Reglas para elegir el nombre de una variable

- El nombre tiene que ser simple, claro y significativo, debe tener sentido y hacer referencia al valor que deberemos guardar en ella. Esto facilita la legibilidad del algoritmo a cualquier persona que tenga que leerlo y a nosotros mismos si, pasado un tiempo, tenemos que modificarlo.
- Las variables tienen que declararse antes de usarlas, así el algoritmo sabe a qué nos estamos refiriendo.
- En el nombre podemos usar letras y números.
- El nombre no puede empezar por un número
- El nombre no puede tener espacios en blanco.
- No podemos usar como nombre palabras reservadas de la notación (**var**, **const**, **fvar**, etc.).
- Las mismas reglas se usan para elegir los nombres de las constantes, las funciones, los algoritmos, etc.

Esta convención de nomenclatura es adoptada generalmente por todos los lenguajes de programación como un hábito que los buenos programadores siguen.

Serían nombres correctos: *segundosIniciales* o *segundos_iniciales*. E incorrectos o no recomendables: *s* (no se entiende el significado), *s_ini* (poco concreto), *segundos iniciales* (contiene espacios).

¿Cómo podemos asignar un valor a una variable?

En un algoritmo ya sabemos cómo indicar qué variables utilizaremos y de qué tipo son. Para poder usar una variable en un algoritmo, necesitamos poder darle (asignarle) un valor, es decir, saber cómo guardar un valor en la caja para poderlo usar en el algoritmo cuando nos haga falta recordarlo.

Fijaos si es importante poder hacer esto que la acción de asignar un valor a una variable es una operación básica y fundamental sin la que no se puede pensar ningún algoritmo. La asignación se representa con el símbolo **:=** que se lee como "toma como valor".

Evidentemente, a una variable solo se le puede asignar un valor que sea del mismo tipo que el suyo.

Concepto clave

Una **variable** es de hecho espacio de la memoria del ordenador que almacena valores. Ahora, sin embargo, no nos preocupa cómo se implementa físicamente una variable en la memoria del ordenador, sino cómo se puede declarar y usar en un algoritmo para guardar los datos que se tengan que tratar.

Las variables son un elemento clave en la algorítmica, porque nos permiten tratar con los datos: podemos guardar valores y realizar operaciones con ellas, intervienen en la toma de decisiones cuando tenemos que decidir entre una alternativa u otra, y nos ayudan a determinar cuándo hay que detener una iteración, como veremos más adelante.

Una variable se declara indicando sus tres elementos:

1. **Nombre**
2. **Tipo de dato que puede guardar**
3. **Valor que tiene en un momento dado**

Recordad que cuando un valor no tiene que variar durante las operaciones del algoritmo se utiliza una **constante** que se identifica con el nombre y el valor concreto. Para asignar un valor determinado a una variable se utiliza la asignación **:=** que es la acción más básica que se puede hacer en un algoritmo. Ya hemos visto que el símbolo de asignación se utiliza también para indicar (asignar) el valor de una constante.

Vemos en este vídeo ("The box variable activity" 7 min. aprox.) las variables "en acción". El vídeo utiliza un ejemplo en lenguaje Python, que tiene una sintaxis algo diferente de la notación algorítmica que acabamos de describir; por ejemplo, para asignar en vez de usar el símbolo **:=** utiliza el símbolo **=** pero ilustra muy bien cómo se opera con variables.

```
color1 := "rojo"
color2 := "verde"

temp := color1
color1 := color2
color2 := temp;
```

Tipos básicos de datos

En los algoritmos tendremos que manipular datos de tipos muy diversos que deberemos poder representar algorítmicamente de algún modo. Para ello, la algorítmica tiene predefinidos unos tipos básicos de datos con los que podemos representar los datos con los que se tenga que trabajar para solucionar el problema (datos de entrada y datos de salida, y también datos intermedios que deban utilizarse para hacer los cálculos).

Los tipos básicos nos permiten representar números (**entero** y **real**) y textos (**carácter** y texto, o **cadena de caracteres**).

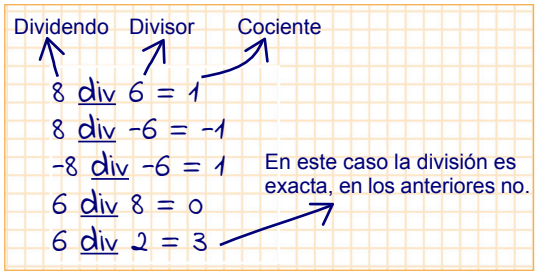
Sintaxis para declarar los tipos básicos

```
var
   nombre1 : entero;
   nombre2 : real;
   nombre3 : caracter;
   nombre4 : cadena;
fvar
```

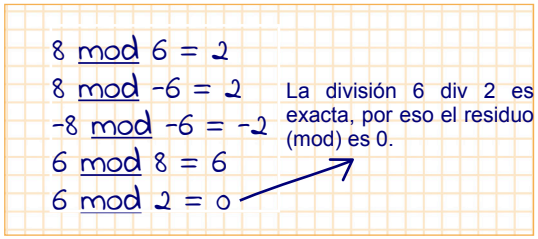
El tipo **ENTERO** representa un número positivo o negativo sin parte decimal, por ejemplo, el 103.

Los enteros se pueden **sumar**, **restar**, **multiplicar** y **dividir**, también les podemos **cambiar el signo**. La notación algorítmica tiene operadores que permiten hacer estas operaciones. Pero atención, algorítmicamente, estas operaciones que podemos hacer con los enteros (que son las que después, en un lenguaje de programación, una máquina las sabrá interpretar) son **operaciones internas**, lo que significa que el resultado tiene que ser del mismo tipo que los operandos, en definitiva, que dan como resultado un entero.

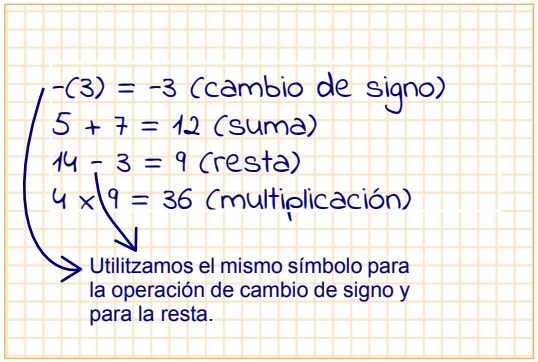
Así pues, la división que podemos hacer con enteros es la división entera, que se representa con el operador **div**, que nos devuelve el cociente de la división entre dos enteros, sin decimales. La división no es exacta en algunos casos.



Y, ¿cómo podemos saber si la división es exacta o no? Conociendo el residuo: si es 0, la división es exacta. Además, tenemos un operador para calcular el módulo (residuo) de una división entera: **mod.**

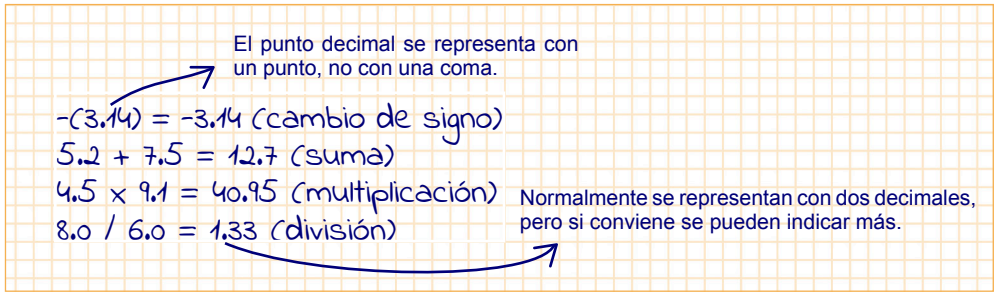


El resto de operaciones aritméticas no tienen mucho misterio algorítmicamente hablando, ya que funcionan tal como las utilizamos matemáticamente (como lo hacemos con una calculadora).



El tipo **REAL** representa un número positivo o negativo con punto decimal, por ejemplo, 2.5.

Los reales también se pueden sumar, restar, multiplicar y dividir, y también les podemos cambiar el signo. La notación algorítmica tiene operadores que permiten hacer estas operaciones (**+**, **-**, **X**, **/**). En este caso, algorítmicamente, estas operaciones algebraicas son tal como las entendemos matemáticamente.



Ejemplo

Ahora sí que ya podemos resolver nuestro problema. ¿Recordáis? Pasar los 103 segundos a minutos y segundos.

¿Cómo representamos los minutos y segundos en algorítmica?

```
var
  segundos_iniciales : entero;
  minutos : entero;
  segundos_restantes : entero;
fvar

segundos_iniciales := 103;
minutos := (segundos_iniciales div 60);
segundos_restantes := segundos_iniciales mod 60;
```

La variable *segundos_iniciales* toma como valor 103.

Las palabras que dan nombre a los operadores también son palabras reservadas del lenguaje, por eso se indican en negrita en los algoritmos (o subrayadas si escribimos a mano).

Otros posibles nombres correctos: *segundosIniciales*, *s_ini*, aunque este es un poco impreciso y podría llevar a confusión. Y sería poco recomendable el nombre *s* porque no aporta significado sobre el sentido de la variable.

Sería incorrecto: *segundos iniciales* porque contiene espacios.

O también, más sintéticamente:

```
var
  segundos_iniciales, minutos, segundos_restantes : entero
fvar
```

¡Y aquí tenemos el algoritmo que resuelve nuestro problema!

El tipo **CARÁCTER** representa cualquier carácter alfanumérico: una letra, un dígito (cifra), un espacio o un símbolo, por ejemplo: la letra 'a' o el dígito '3' o el signo de interrogación '?'. Para distinguirlos de los números, se escriben normalmente entre comillas simples: 'a', '3', o '@'.

Con los **caracteres** no podemos hacer operaciones aritméticas, obviamente; no podemos los podemos sumar, ni restar, etc., pero sí que los podemos concatenar.

No es lo mismo el número entero 3 que el carácter '3', aunque para nosotros puedan tener el mismo significado.

El tipo **CADENA DE CARACTERES** representa un conjunto de caracteres, un texto. También se escriben entre comillas: "Esto es un texto", "una cadena de caracteres".

La operación de **concatenación** la representamos con el símbolo **+**:
"Esto es un texto" + "," + "una cadena de caracteres" + "." generaría una frase mucho más larga: "Esto es un texto, una cadena de caracteres."

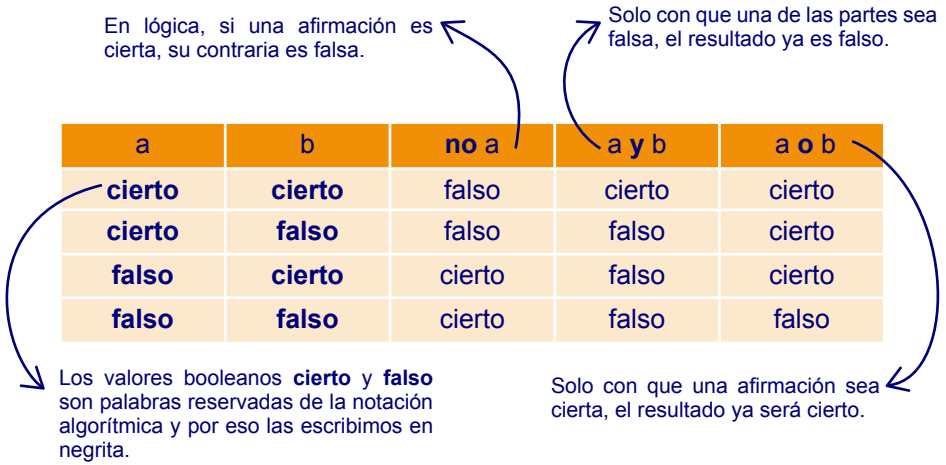
Con las c**adenas de caracteres** tampoco podemos hacer operaciones aritméticas pero sí que las podemos concatenar, unir, para formar frases más largas.

Añadiremos un último tipo básico importantísimo para la algorítmica: el tipo **BOOLEANO**. Aparentemente, es un tipo que puede parecer muy poca cosa porque es un tipo que solo puede tener dos valores: **cierto** y **falso**. Y eso es todo. Pero pese a ser un tipo peculiar es esencial, por ejemplo, para poder tomar decisiones en un algoritmo

En lógica, este tipo se utiliza para indicar la certeza o falsedad de un hecho (de un enunciado, diríamos en lógica).

Con los **valores booleanos** se pueden realizar operaciones, o sea, combinar diferentes condiciones ciertas o falsas entre sí y ver si el resultado es cierto o falso. Los valores booleanos se combinan entre ellos de acuerdo con la lógica binaria y con tres operadores lógicos: **no** (negación), **y** (se lee como "y lógico") y **o** ("o lógico").

Podemos ver todas las combinaciones de los valores **cierto** y **falso** con estas operaciones en una única tabla que se conoce con el nombre de la **tabla de verdad**:



a	b	**no** a	a **y** b	a **o** b
cierto	**cierto**	falso	cierto	cierto
cierto	**falso**	falso	falso	cierto
falso	**cierto**	cierto	falso	cierto
falso	**falso**	cierto	falso	falso

Vemos en este vídeo ("Boolean: comnining keywords" 4 min. aprox.) una aplicación práctica de estas operaciones.

El tipo booleano tiene su origen en el álgebra de Boole, ideada por el matemático Georges Boole en el siglo XIX. Trabaja con estos dos valores lógicos binarios, **verdadero** y **falso**, que a menudo están representados como **1** y **0**, respectivamente, y que son la base de la computación.

Mezclando tipos

Seguramente, para resolver problemas en algunos casos habrá que comparar valores o saber si un número es mayor que el otro, por ejemplo. El tipo booleano permite ampliar el abanico de operaciones de los otros tipos elementales con **operaciones externas** (las llamamos externas porque dan como resultado un valor de un tipo diferente, en este caso booleano).

5 > 3 es cierto
5 < 3 es falso
5 = 3 es falso
5 ≠ 3 es cierto
5 ≥ 3 es cierto
5 ≤ 3 es falso

Aunque parezca raro, estas operaciones también se pueden aplicar a los caracteres e incluso a los booleanos. Esto sucede porque los caracteres están ordenados alfabéticamente: la 'a' está antes que la 'b' y así sucesivamente. Por tanto, cuando decimos que 'a' es más pequeño que 'b' en realidad queremos decir que la letra 'a' va antes que la 'b' en el alfabeto.

'a' = 'b' es falso
'a' < 'b' es cierto
'a' > 'b' es falso pero 'A' < 'b' es cierto
cierto > **falso**

Por convención, las minúsculas son mayores que las mayúsculas.

Por convención, también el valor **cierto** se considera mayor que el valor **falso**.

Y, por extensión, también podemos hacer lo mismo con las cadenas (evaluando carácter a carácter desde el primero hasta el último y en el mismo orden en el que están escritas:

"abcd" > "abcz" es falso
"Hola" < "adiós" es cierto

La 'd' es más pequeña que la 'z'.

Las minúsculas se consideran "mayores" que las mayúsculas.

¡Pero cuidado! Cuando hacemos cálculos y comparaciones con los tipos elementales, hay que recordar siempre que "No se pueden sumar peras con manzanas", en definitiva, que podemos sumar dos enteros, multiplicar dos reales, comparar dos letras, pero no podemos mezclar en una misma operación valores de tipos diferentes.

Incorrecto	Correcto
5 + 7.3	5.0 + 7.3
5.0 + 4	5 + 4
"el resultado es el número:" + 1	"el resultado es el número:" + "1"

Funciones de conversión de tipos

A veces, sin embargo, es necesario poder mezclar datos de diferentes tipos. Por ello, la algorítmica dispone de una lista de funciones predefinidas de conversión de tipo, que cambian el tipo de un dato a otro tipo.

El ASCII es un juego de caracteres que asigna valores numéricos (del 0 al 127, 7 bits de longitud) a las letras, las cifras y los signos de puntuación. Casi todos los sistemas informáticos utilizan el código ASCII para representar textos.

realAEntero	Convierte un valor real a entero, dejando solo la parte entera del valor real. **realAEntero** (3.5) devuelve el valor entero 3.
enteroAReal	Convierte un valor entero a real, agregando 0 a la parte decimal. **enteroAReal** (3) devuelve 3.0.
caracterAEntero	Convierte un carácter a entero. Este entero es el que le corresponde al carácter según el sistema de codificación ASCII. **caracterAEntero** ('A') da como resultado 65, que es el número con el que en codificación ASCII se representa la letra A mayúscula.
enteroACaracter	Convierte un entero en un carácter aplicando la tabla de codificación ASCII. **enteroACaracter** (65) devuelve el carácter 'A'.
enteroACadena	Convierte un entero en una cadena de caracteres. **enteroACadena** (4) da como resultado '4'.

Marcamos las funciones de conversión en negrita porque son palabras reservadas de la notación. Esto significa que no podemos usarlas para nombres de variables o funciones que inventamos nosotros porque el equipo se confundiría al interpretarlas.

Fijaos que para un ordenador no es lo mismo el número entero 4 que el carácter '4'. Nosotros, al ver un 4, sabemos interpretar que es el número 4, pero para un ordenador es un número solo si no lleva comillas.

Expresiones

A las instrucciones para realizar cálculos las llamamos expresiones. Normalmente, se trata de operaciones con números (aritméticas). Y las podemos indicar con los operadores asociados a cada tipo: **+**, **-**, **div**, etc. Pero también podemos construir expresiones con booleanos, caracteres o cadenas.

Podemos construir expresiones con valores (por ejemplo, 3 + 5) o con variables (por ejemplo, *lluviasEnero* + *lluviasFebrero* + *lluviasMarzo* = *lluviasPrimerTrimestre* donde las cuatro variables fueran de tipo entero).

Eso sí, es importante vigilar que los tipos de valores o de variables que usemos en la expresión sean correctos, por ejemplo, para sumar hacen falta dos enteros o dos reales, porque la suma necesita al menos dos sumandos. Y no podríamos sumar dos booleanos. Sería operaciones incorrectas: no(7), 5+**cierto**, 6/2.

Reglas para calcular expresiones

- Cuando en una expresión hay varios operadores, se empieza a evaluarlos siguiendo la siguiente tabla, donde el operador más prioritario es el más arriba:
- Las operaciones que hay entre paréntesis tienen prioridad respecto a las de fuera.

cambio de signo	no				
x	/	div	mod		
+	-				
=	≠	<	>	≤	≥
y					
o					

- Los operdores de la misma fila, tienen la misma precedencia. La precedencia determina el orden en que se aplican los operadores cuando se evalua la expresión, qué operación haremos antes y cuál después. Se evalua primero la que se encuentra más a la izquierda de la expresión concreta que estemos evalulando:

2.0 + 8.0 / 2.0 +10.0 x 5.0
2.0 + 4.0 + 10.0 x 5.0
2.0 + 4.0 + 50.0
6.0 + 50.0
56.0

Ejemplos de expresiones

Estas tres expresiones NO serán correctas:

```
3 x 5 cierto
```
→ No tiene ningún sentido, podríamos calcular 3x5 y daría 15, pero 15 verdadero no tiene ningún sentido, no hay ninguna operación que vincule el entero 15 y el booleano **cierto**.

```
const
  const1 : real := 1.0;
fconst
var
 var1: entero;
fvar
const1 x entero1
```
→ Es incorrecto porque intentamos multiplicar un real con un entero.

```
var
  r1, r2: real;
fvar
(r1/r2) 4.5
```
→ 1.º: (r1/r2) da como resultado un real.
2.º: real 4.5 no significa nada, ¡falta el operador! El ordenador no sabe interpretar que si no hay ningún operador es una multiplicación.

Estas cuatro expresiones serían correctas:

```
var
 c, d, e : entero;
fvar
c div d x e
```
→ Primero calculamos *c* div *d* y da como resultado un entero, después lo multiplicamos por *e* y da otro entero como resultado final.

```
5 = 4 - -(1)
```
→ Calculamos:
1r: -(1) da -1
2n: - -1 da +1
3r: 4 +1 da 5
4t: 5 = 5 da **cierto**

```
3.5 > 2.8
```
→ Daría como resultado **cierto**.

```
'a' < 'A'
```
→ Daría **falso** porque, según la codificación ASCII, las minúsculas son mayores que las mayúsculas.

Conceptos clave

Los **tipos básicos de datos** permiten representar datos simples. También se denominan tipos elementales.

En algorítmica, los tipos básicos para representar números son el tipo **entero** y el tipo **real**, y para representar los textos el tipo **carácter** y el tipo **cadena** de tipo de caracteres (*string* en inglés).

Con los tipos numéricos se pueden realizar las **operaciones** habituales de la aritmética matemática: **suma, resta, multiplicación y división**, y también se les puede **cambiar el signo** (de positivo a negativo o viceversa), siempre teniendo en cuenta que no se pueden mezclar valores de diferentes tipos en una misma operación (son **operaciones internas**, esto es que dan como resultado un elemento del mismo tipo). Por ello, en el caso de los enteros, utilizamos la división de enterors (**div**) y el módulo (**mod**) que nos devuelve el residuo de una división.

Estos operadores son elementos predefinidos de la notación algorítmica que seguimos y, por lo tanto, se pueden utilizar en algoritmos. Por esto están escritos en negrita, son palabras reservadas de la notación algorítmica.

Dado que no es posible mezclar valores de diferentes tipos en una misma operación, a veces necesitamos utilizar las **funciones de conversión de tipo** que nos permiten pasar un valor de un tipo a otro: de entero a carácter, a cadena o a real, de carácter a entero cuando esto tenga sentido, y de real a entero. Son funciones predefinidas en la notación algorítmica, por lo que no hay que preocuparse por cómo funcionan y podemos usarlas directamente. Por eso también las escribimos en negrita.

Un tipo imprescindible y de los más utilizados en la algorítmica es el **booleano**, que solo tiene dos valores posibles: **cierto** y **falso**. Su origen está en el álgebra de Boole. También se denomina tipo lógico. Con los booleanos podemos hacer tres operaciones lógicas: **no** (negación), **y** (conjunción) y **o** (disyunción). La tabla de verdad detalla el resultado de estas operaciones.

El tipo booleano es imprescindible en algorítmica porque es necesario para decidir si ejecutar unas instrucciones u otras, y también para saber hasta cuándo hay que estar repitiendo unas instrucciones concretas, por ejemplo.

También nos permite ampliar el abanico de operaciones de los otros tipos elementales con **operaciones externas** (las llamamos externas porque dan como resultado un valor de un tipo diferente, en este caso booleano): podemos comparar números, saber si uno es mayor que el otro, si es igual, si es menor o si es diferente.

A todas estas instrucciones con las que realizamos operaciones diversas, en algorítmica las llamamos **expresiones**.

Comunicación con el exterior: entrada/salida

Un algoritmo es un conjunto de instrucciones que resuelve un problema. Para hacerlo le hacen falta datos, son los datos de entrada. Y, una vez resuelto el problema, debe comunicar el resultado, que son los datos de salida.

¿Cómo podemos obtener los datos de entrada del algoritmo?

El algoritmo puede conseguir los datos que necesita para resolver el problema de distintos modos:

- Solicitarlos al usuario por pantalla y leerlos desde el teclado con instrucciones del algoritmo como una instrucción del algoritmo.
- Escribirlos directamente en el algoritmo como constantes (a esto, en inglés se lo conoce como *hard code*).
- Leerlos de un fichero en el que estén almacenados.

Para solicitar datos al usuario, necesitamos funciones de entrada/salida que nos permitan escribir por la pantalla lo que le queremos pedir y leer del teclado lo que el usuario nos escribe. Por ello, los lenguajes disponen de funciones predefinidas. Algorítmicamente, las indicamos así:

```
escribirPantalla (valor)
v:= leerTeclado();
```

Visualiza por pantalla el valor que le pasamos como parámetro.

Lee del teclado un valor y lo guarda en la variable.

La opción de leer datos desde ficheros no la veremos, por ahora

¿Cómo mostrar los resultados?

La función de escritura por pantalla también nos servirá para mostrar al usuario los resultados de nuestro algoritmo. Igualmente, podríamos dejar los resultados en un fichero, pero esta opción no la veremos, por ahora.

Habitualmente, hay distintas funciones de lectura y escritura para cada tipo de dato, algorítmicamente lo simplificamos en una sola para todos los tipos. Cuando programamos en un lenguaje concreto debemos estar atentos a sus especificidades y elegir la que convenga según el caso.

Ejemplo

Retomamos ahora el algoritmo que transformaba los segundos de una llamada telefónica a minutos y segundos. ¿Lo recordáis?

```
var
  segundos_iniciales : entero;
  minutos : entero;
  segundos_restantes : entero;
fvar

segundos_iniciales := 103;
minutos := (segundos_iniciales div 60);
segundos_restantes := segundos_iniciales mod 60;
```

Hemos utilizado la técnica de *hard code* para asignar el dato de entrada al algoritmo.

O también, más sintéticamente:

```
var
  segundos_iniciales, minutos, segundos_restantes : entero
fvar
```

Ejemplo

Ahora generalicémoslo para cualquier valor de entrada que pediremos al usuario por pantalla:

```
var
  segundos_iniciales : entero;
  minutos : entero;
  segundos_restantes : entero;
fvar

escribirPantalla("Indica los segundos a convertir por favor:");
segundos_iniciales := leerTeclado();
minutos := (segundos_iniciales div 60);
segundos_restantes := segundos_iniciales mod 60;
escribirPantalla("El resultado es: "+ enteroACadena(minutos) + "minutos y" + enteroACadena(segundos_restantes) + "segundos");
```

Otro posible nombre correcto sería *segundosIniciales*.

Serían poco recomendables *s_ini* porque es un poco vago y podría llevar a confusión y *s* porque no aprota significado al sentido de la variable.

Sería incorrecto *segundos iniciales* porque contiene espacios.

Escribimos por pantalla esta frase.

Leemos de teclado el dato de entrada.

Variables que contiene los segundos restantes.

Concatenamos los valores y se mostrará por pantalla el siguiente texto: "El resultado es: 1 minutos y 43 segundos."

El operador de asignación también es un símbolo reservado de la notación algorítmica.

Variable que contiene los minutos resultantes.

Aquí habría que escribir "minutos" o "minuto" dependiendo de si el valor de la variable *minutos* es 1 o más. Aprenderemos a hacerlo más adelante con las estructuras algorítmicas. Lo mismo ocurrirá en el caso de la variable *segundos_restantes*.

O también, más sintéticamente:

```
var
  segundosIniciales, minutos, segundosRestantes : entero;
fvar
```

Concepto clave

Las **funciones predefinidas de entrada/salida** permiten que el algoritmo reciba los datos que necesita para resolver el problema y también que muestre los resultados. Habitualmente, los lenguajes de programación disponen de diversas funciones para cada tipo de dato básico: para leer enteros, para leer reales, para escribir un carácter, etc.

Otros modos de obtener los datos de entrada son definirlos dentro del algoritmo o programa (*hard code*) o leerlos de un fichero externo. Igualmente, el resultado se podría almacenar en un fichero en lugar de mostrarse por pantalla. Dependiendo del caso, tendrá sentido hacerlo de un modo u otro.

¿Cómo trabajamos con los tipos de datos básicos en JS?

La gramática del lenguaje JS tiene sus peculiaridades en relación con los tipos de datos. Por ejemplo, hay dos modos de declarar variables (**let** y **var**). Para empezar a programar, utilizaremos las opciones más sencillas y más recomendables atendiendo a lo que hemos visto algorítmicamente.

Sintaxis para declarar los tipos básicos de datos en JS

```
var nombreVariable;
```

En JS no hay que indicar el tipo de la variable en su declaración.

Una característica del JS es que es un lenguaje "débilmente tipado", esto significa que no es necesario declarar el tipo de una variable cuando se define, el tipo se asume dinámicamente en el momento en que se le asigna un valor, o sea, que cuando damos un valor a la variable, el programa sabe de qué tipo es la variable.

De hecho, en JS ni siquiera es obligatorio declarar variables antes de usarlas, aunque es una práctica nada recomendable.

Ejemplo

```
1 //Variables
2 var segundos_iniciales = 103;
3 var minutos;
4 var segundos_restantes;
5 //Constante
6 const segundosMinuto = 60;
7 //Línea de comentario. No se ejecutará.
8
```

JS es *case-sensitive*, esto significa que distingue entre mayúsculas y minúsculas, por lo que el nombre "segundos_iniciales" diferirá de "segundos_Iniciales".

Así se definen las constantes en JS.

A pesar de que no es obligatorio, se considera una buena práctica que las instrucciones terminen con punto y coma. Aunque desde la consola nos funcionará habitualmente sin punto y coma, pensad que no es el único lugar desde el que se ejecuta JS, por lo que es mejor ponerlo siempre.

Es una muy buena costumbre comentar el código, facilita la legibilidad y las posteriores modificaciones. Usamos la doble barra para indicar que es una línea de comentarios y que, por lo tanto, no debe ejecutarse.

¿Cómo se asignan valores a variables y constantes?

Para asignar valores a variables o constantes usaremos el símbolo **=**.

```
nombreVariable = valor;
```

Así pues, no podemos usar el símbolo **=** para comparar valores porque sería confuso. Para comparar valores se utiliza el símbolo **==**.

Tipos de datos básicos

- El JS trabaja solo con tres tipos básicos de datos: **Number**, **String** y **Boolean**. Por lo tanto, no distingue entre entero y real ni entre carácter y cadena de caracteres. Esto a menudo simplifica los cálculos.

- En el caso del **tipo Number**, los símbolos de los operadores aritméticos son: **+**, **-**, **x**, **/** y **%**, que corresponden al operador **mod** algorítmico (calcula el módulo de una división). La coma decimal se representa con un "**.**". Los operadores de comparación son: **==** (igual), **!=** (diferente), **>** (mayor), **<** (menor), **<=** (menor o igual) y **>=** (mayor o igual).

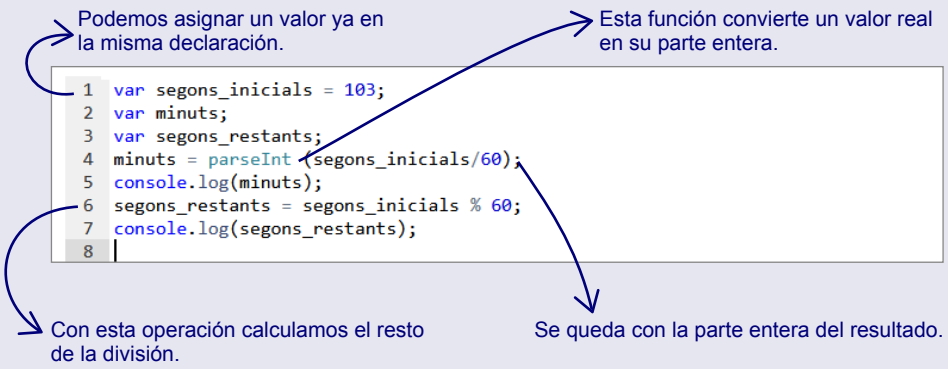
  Y contamos también con las siguientes funciones de conversión de tipo: **parseInt()** y **parseFloat()**, que convierten un valor en un entero o en un real, respectivamente.

- En el caso del **tipo Boolean**, los símbolos de los operadores son: **&&** (la Y lógica), **||** (la O lógica) y **!** para la negación. Y los valores **true** y **false** se escriben en minúscula.

- El **tipo String** se representa entre comillas dobles, por ejemplo: "Joan". Si concatenamos un Number con un String con el operador de concatenación +, JS realiza la conversión automática del entero a String, por eso no es necesario tener funciones predefinidas de número a carácter.

Ejemplo

Siguiendo con el ejemplo anterior, aquí tenemos su codificación en JS.



En JS se utiliza la función **console.log()** para mostrar por pantalla el valor de una variable o un literal. Y la función **window.prompt()** para pedir y leer un valor desde la pantalla.

¿Qué pasaría si...?

Combinando los tres tipos, podemos encontrarnos con resultados un poco sorprendentes:

Ejemplo

Convierte automáticamente el número en *string*.

```
1  // ... comparamos diferentes tipos
2  var numberNum = 1;
3  var stringNum = "1";
4  console.log (numberNum == stringNum); //retorna true !!!
5
6  // ... si concatenamos tipos diferentes
7  var numberNum = 1;
8  var stringNum = "1";
9  var concatena = stringNum + numberNum;
10 console.log (concatena); // escribe por pantalla "11"
11
```

Igualmente aquí, antes de concatenar JS, convierte automáticamente la variable numérica en *string*.

La práctica y el conocimiento de un lenguaje de programación concreto nos permitirán conocer estos comportamientos automáticos del lenguaje que inicialmente pueden parecer extraños. Recordad que cada lenguaje tiene sus particularidades e implementa de una manera distinta los principios algorítmicos que son más universales. Algo similar sucede con los idiomas.

La fase 4 de Implementar consiste precisamente en encontrar las instrucciones, operadores y funciones concretas del lenguaje más adecuadas para expresar el algoritmo de resolución que tenemos pensado, teniendo en cuenta las particularidades concretas de cada lenguaje. No se trata, en absoluto, de una simple traducción de la notación algorítmica al lenguaje de programación, sino más bien de la concreción de las ideas algorítmicas de resolución del problema en el lenguaje de programación escogido. Debería ser un paso no demasiado complicado puesto que todos los lenguajes estructurados comparten unas características fundamentales comunes. Más adelante, la destreza en un lenguaje de programación concreto nos permitirá sacar el mejor partido al lenguaje para hacer programas más eficientes.

Encontraréis una explicación detallada del funcionamiento de los tipos básicos en JavaScript en la guía de JavaScript de Mozilla.

Podéis probar estos códigos en la web de PythonTutor.