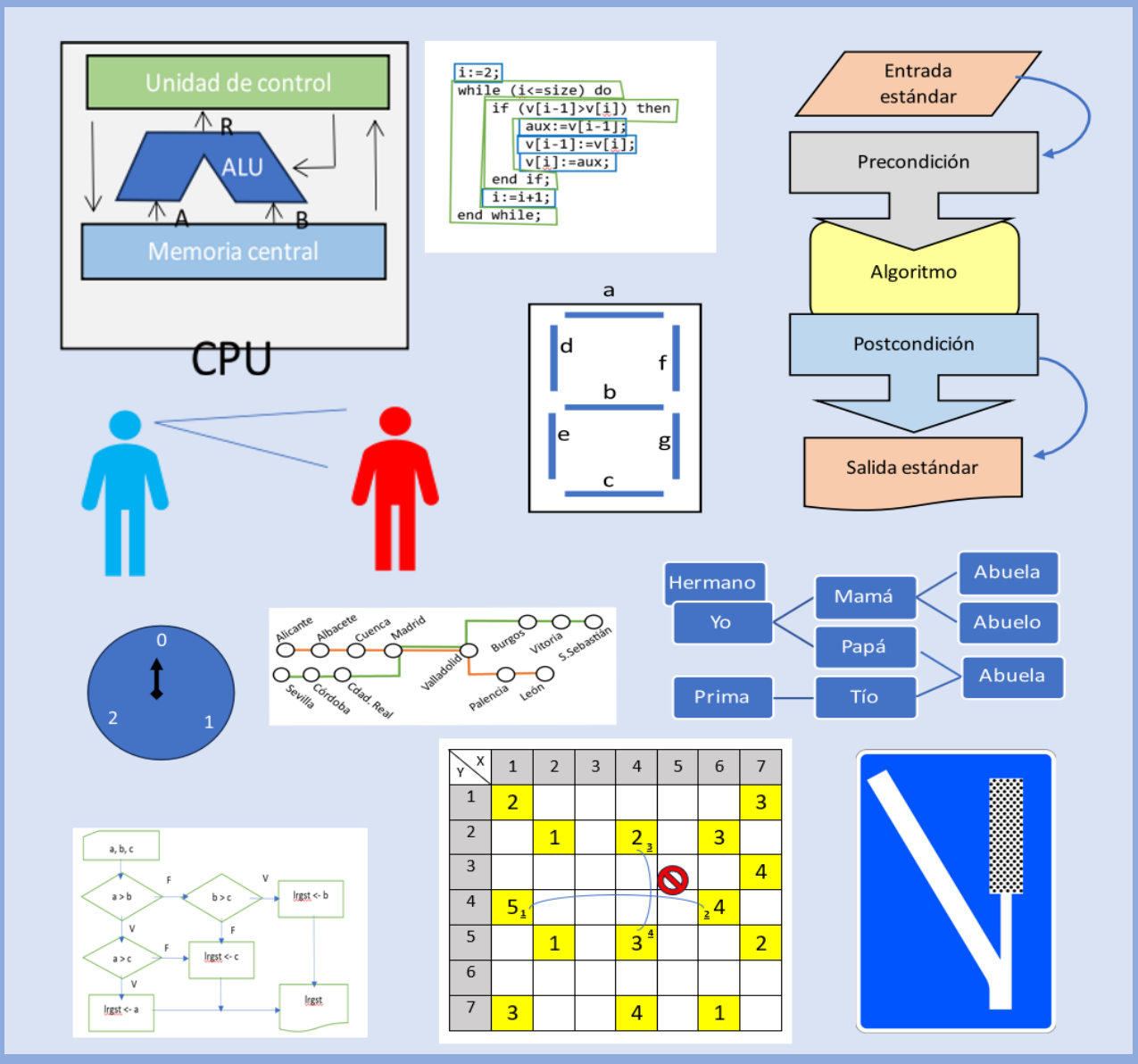
BASES DE LA PROGRAMACIÓN

(BLOQUES A-B)

A: INTR. AL DISEÑO DE PROGRAMAS – B: INTR. A LA CIENCIA DE DATOS

Furro Computero (Zorrozoaga)



Bases de la Programación

(Introducción al Diseño de Programas y a la Ciencia de Datos)

Guía básica para discursar con un ordenador.

Autor: Furro Computero

Índice de contenidos

PRÓLOGO: La Informática

La palabra “Informática” proviene de la combinación de las palabras “Información” y “Automática”, es por ende, la ciencia que se encarga del tratamiento automático de la información. Detrás de esta definición, yace un motor invisible que le da un propósito a la Informática: la Programación. Pues bien, de cualquier ordenador, podemos distinguir sus dos componentes esenciales: el hardware y el software.

El hardware (“componentes duros” del inglés) se corresponde con los componentes físicos del ordenador, aquellos que son tangibles, que se pueden tocar, como el CPU, la memoria, el teclado o la pantalla. Por otra parte, el software (“componentes blandos” del inglés) se corresponde con las instrucciones que sigue un ordenador para poder ejecutar los programas y el sistema operativo, es decir, una serie de datos e información intangible que viaja por los buses entre componentes físicos y le dictaminan al hardware qué debe realizar.

De esta forma, el hardware es el cuerpo que contiene el conocimiento mientras que el software es el lenguaje del pensamiento que permite articularlo. Son como cuerpo y alma, respectivamente, uno no puede sobrevivir sin el otro; el hardware necesita software (instrucciones) para operar, y el software necesita hardware (medio físico) para ejecutarse.

Programar no es tan simple como dar órdenes a una máquina, sino un arte, el arte de traducir la lógica humana al dominio de los circuitos electrónicos que posee el hardware. Es el proceso creativo y riguroso mediante el cual diseñamos soluciones a problemas reales, estructuramos el caos de la información y transformamos ideas abstractas en software, instrucciones que el hardware sea capaz de interpretar y ejecutar. Sin la Programación, la "Automática" sería una estructura inerte; con ella, se convierte en una herramienta capaz de explorar y expandir los límites de la creatividad y capacidad humana.

En resumen, la programación es el ejercicio de entablar una conversación con el ordenador para darle instrucciones de qué debe realizar para resolver un problema.

Este prólogo se limita a explorar la Informática como un puente de dos extremos donde la Programación es uno de los dos, siendo el otro la Tecnología de Computadores. Entender la Programación es comprender el tejido del hoy, pues en cada línea de código se escribe el presente y el futuro de nuestra Era Digital. El otro extremo no se verá en este libro, pero sí es importante recalcar que ambos saberes son concurrentes, y que sin uno de ellos el otro carece de existencia.

BLOQUE A: INTR. AL DISEÑO DE PROGRAMAS

0. Discursar con un ordenador: Programas

Los discursos son muy frecuentes en el día a día, refiriéndome a discurso como cualquier acción, por muy simple que esta sea, de conversación utilizando cualquier medio de comunicación ya sea oral u escrito. Cualquier narración, descripción, exposición, argumentación, instrucción, etc. en cualquier medio como la radio, el diálogo, la televisión, el periódico, la revista o el libro son ejemplos de discursos.

En nuestro caso, para entablar cualquier tipo de discurso que queramos, solamente se necesitaría un medio como el habla o el papel y bolígrafo y un idioma. Por ejemplo, para escribir esta guía de programación, hemos necesitado de un editor de documentos de mi ordenador y de lengua castellana.

Sin embargo, este ordenador no escribe esta guía por la magia de darle a las teclas correctas para formar oraciones ni las maqueta para que correspondan con palabras claves, título del capítulo, título de la sección, etc. de la nada, sino porque hay un **programa** que me permite realizar estas acciones.

Pero vayamos al principio, a la pregunta primera que nos debemos hacer. ¿Qué es un programa? Según la doctrina informática, se define como un **conjunto de instrucciones basadas en un lenguaje de programación** específico que un ordenador usa para **resolver un problema determinado**, es decir, una serie de pasos a seguir que se asemeja al **discurso instructivo** como cuando por ejemplo se quiere hacer una receta de un libro de cocinas.

Sin embargo, esto no es del todo cierto. Un programa no tiene por qué ser algo que simplemente siga una serie de pasos como si fuera una receta para hacer cosas muy específicas como calcular la raíz cuadrada, buscar un objeto de la lista o rellenar celdas de una base de datos. Un programa puede cumplir con un **propósito** más **general** como por ejemplo un editor de documentos, un juego de estrategia o un software de edición de multimedia. En estos programas, no existe un único rompecabezas o proceso a resolver, sino que este debe cumplir con su cometido resolviendo tantos rompecabezas como se precise.

Poniendo de ejemplo el juego de estrategia, debe reunir un montón de características como los escenarios, la historia, los niveles, los personajes, la jugabilidad, etc. Sus problemas concretos o rompecabezas como me gusta llamarlos ocupan un buen fragmento de código que en conjunto todos forman el programa. Necesitamos una parte de código que grafique en pantalla los escenarios, otra que configure los movimientos del personaje, otra que asigne la puntuación del personaje por enemigo derrotado, etc.

Ese **conjunto de instrucciones definido, ordenado y finito** que soluciona un problema, realizando **cómputos** o procesando **datos** determinados es lo que comúnmente denominamos **algoritmo**. Luego, el programa que gestiona el juego de estrategia debe gestionar distintos algoritmos para ejecutar correctamente todos los aspectos del juego.

Todavía no desarrollaremos el concepto de algoritmo, este se hará en el siguiente tema, pues me gustaría centrarme más en el mero funcionamiento de un ordenador.

Pregunta, ¿Cómo entabla el ordenador un discurso? Respuesta, por medio de programas. ¿Cómo los llega a comprender el ordenador? Vamos a abstraernos de ese proceso y nos lo vamos a imaginar como si el ordenador fuera un gólem del cual tú metes un trozo de pergamino en la boca con datos iniciales y obtienes el resultado final de la tarea que le encomendaste. El gólem tiene una peculiaridad y es que sólo es capaz de comprender 0's y 1's, es decir, sistema binario, pues es su lengua materna.

Por suerte, este gólem también sabe distintos **lenguajes de programación** y es capaz de traducir desde esos lenguajes al **código máquina** binario que conoce. Así mismo, para comprender básicamente cómo funciona la programación, no necesitamos saber cómo se traduce un programa de cualquier lenguaje a código máquina, de la misma forma que no necesitamos saber todo lo que hay dentro de un automóvil para conducir.

Ese paso llamado **compilación** lo percibiremos a partir de ahora como una **caja negra** donde asumiremos que ese proceso es autónomo y que esta caja recibe datos de entrada (input) y envía datos de salida (output), o también simplemente transforman el dato (in&output)

La ventaja de utilizar un lenguaje de programación y no el código máquina, es que podemos conocer a la perfección lo que realiza el programa leyendo únicamente su código, ya que **el lenguaje de programación se asemeja al lenguaje natural** y es totalmente inteligible. Veamos el siguiente programa escrito en lenguaje Pascal:

`program de_24h_a_12h (INPUT, OUTPUT);`	Título del programa
`{Este programa lee un número entero que simboliza una hora en formato 24h como 0835 o 2050 e imprime en pantalla su correspondiente en formato 12h}`	Descripción o especificación del programa
`var` `Hora_24h, Horas, Minutos: Integer;`	Declaración de variables

`begin`
`Write ('Escriba la hora en formato 24h: ');`
`Read (Hora_24h);`
`Horas := Hora_24h div 100;`
`Minutos := Hora_24h mod 100;`
`Write ('La hora en formato 12h es: ');`
`if (Horas = 0) then`
`  Write ('12:', Minutos, ' a.m')`
`else if (Horas > 0) and (Horas < 12) then`
`  Write (Horas, ':', Minutos, ' a.m')`
`else if (Horas = 12) then`
`  Write ('12:', Minutos, ' p.m')`
`else`
`  Write ((Horas – 12), ':', Minutos, ' p.m');`
`end.`
	Instrucciones del programa o algoritmo

Como podemos observar, no necesitamos apenas conocer las reglas sintácticas ni semánticas del lenguaje de programación para comprender lo que hace el programa, sólo se necesita un **conocimiento mínimo de inglés, de matemáticas y de lógica**. Expresemos lo que hace el programa con nuestras palabras:

<<En primer lugar, aparece una cabecera del programa, que indica el **título** de este, y se titula "de_24h_a_12h", lo que nos da una idea de lo que hace el programa. Además, en este caso se nos indican dos etiquetas llamadas INPUT y OUTPUT. Esto permite al programa tanto **leer del teclado** los datos solicitados como **imprimir**los **en pantalla**. La **habilitación de la E/S** depende del lenguaje de programación, cada uno tendrá su manera peculiar.

Tras el título del programa, nos aparece una **breve descripción** de lo que debe realizar, lo que también se denomina como **especificación**. Esto no es obligatorio, pero nos sirve de gran ayuda a la hora de compartir programas con otros desarrolladores con tal de que se comprenda cómo nosotros lo hemos realizado. Esta descripción aparece a modo de **comentario**, y **el ordenador hará caso omiso** de este.

A continuación, se **declaran las variables** que necesite el programa. Las variables son los **espacios de memoria** principal del ordenador que se reservan para una cantidad de **información** conocida o desconocida de un **tipo de dato** en concreto y modificable a lo largo del desarrollo del programa. Así mismo, necesitamos almacenar en nuestro programa tres datos de tipo entero: el entero que simboliza la hora en formato 24h (p.ej. 1615) y otros dos para indicar las horas y minutos que han pasado (siendo Horas=16 y Minutos=15)

Por último, se ejecuta el **algoritmo** necesario para obtener la hora en su correspondiente formato de 12h. Para ello, se nos pide introducir el valor de Hora_24h por el teclado, un entero. Una vez hecho esto, el algoritmo asignará a Horas sus dos primeras cifras y a Minutos, sus dos últimas. En este ejemplo (1615), como Horas es mayor que 12, ejecutará la línea de código del bloque IF correspondiente, restando 12 a Horas, y señalizando debidamente la hora con la marca "p.m", mostrándose en la pantalla "4:15 p.m">>

La estructura del discurso que sigue nuestro programa es muy similar a la de cualquier texto escrito, ya sea narrativo, expositivo, argumentativo, etc.

Programa	Título	Especificación	Declaración	Algoritmo
Instrucción		Introducción	Listado de inventario	Procedimiento
Narración		Planteamiento	Nudo	Desenlace
Descripción		Introducción	Desarrollo	Conclusión
Argumentación		Tesis	Cuerpo	Conclusión
Exposición		Presentación	Desarrollo	Conclusión

Sin embargo, la principal diferencia entre los textos no instructivos y el programa radica en la localización de su parte nuclear. Mientras que en los textos no instructivos la parte nuclear se encuentra en medio del texto, y el final es una conclusión extraída de los acontecimientos, hechos o argumentos de dicha parte; en el programa y en el texto instructivo, antes de desarrollar su núcleo, se procede a hacer un listado del inventario que se necesita para realizar correctamente la tarea, siendo por una parte una lista de ingredientes y utensilios para una receta de un libro de cocina y por la otra parte la declaración de las variables que se necesiten para la correcta ejecución del programa. Luego, la parte principal del programa es el algoritmo responsable de ejecutar la tarea, el cual se desarrolla tras declarar todas las variables que el programa necesita, de forma similar a cuando se escriben los pasos que debe seguir una receta de cocina después de realizar la lista de ingredientes y utensilios necesarios. Ambos funcionan no sólo de desarrollo, sino también de conclusión.

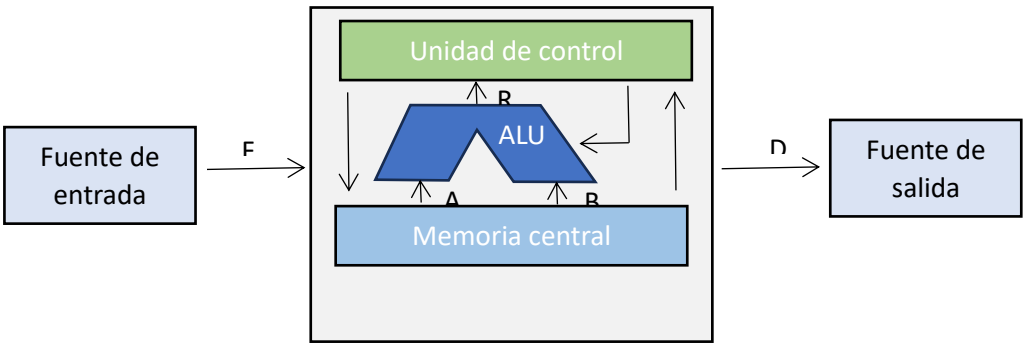
Por otra parte, todos los textos se caracterizan por su introducción, que explica siempre el propósito inicial del texto, ya sea la presentación de la descripción de un lugar, la tesis que se desea defender, un breve precepto de los hechos de una historia o la breve descripción de lo que realiza un programa.

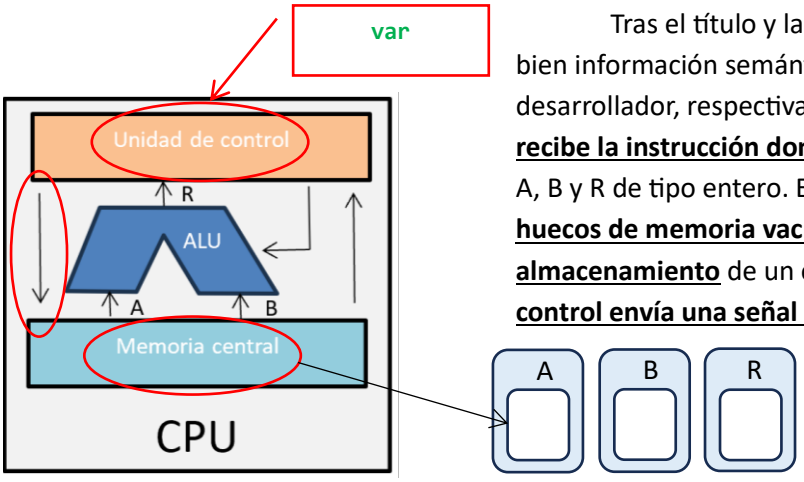
Para terminar este tema, me gustaría explicar un poco más detalladamente cómo el ordenador es capaz de ejecutar un programa a través de los distintos componentes del ordenador (hardware). Para ello, asumimos que nuestro procesador sigue la **arquitectura de Von Neumann**.

Este diseño es bastante más simplificado de lo que sería la arquitectura del computador en realidad, pero lo suficiente como para comprender el mecanismo de la programación. Pondremos como ejemplo otro programa en lenguaje Pascal.

```
program Suma_5 (INPUT, OUTPUT);

{Este programa lee un número entero A por el teclado, al cual se le suma
la variable B, a la que se le asigna el valor 5. El resultado R de la suma
se imprimirá en pantalla}
var
A, B, R: Integer;
begin
Read (A);
B := 5;
R := A + B;
Write (R);
end.
```

Con este programa y el esquema del **CPU**, del inglés, **Unidad Central de Procesamiento** explicaremos cómo el ordenador es capaz de **leer, almacenar, procesar y escribir datos** a través de sus distintos componentes.

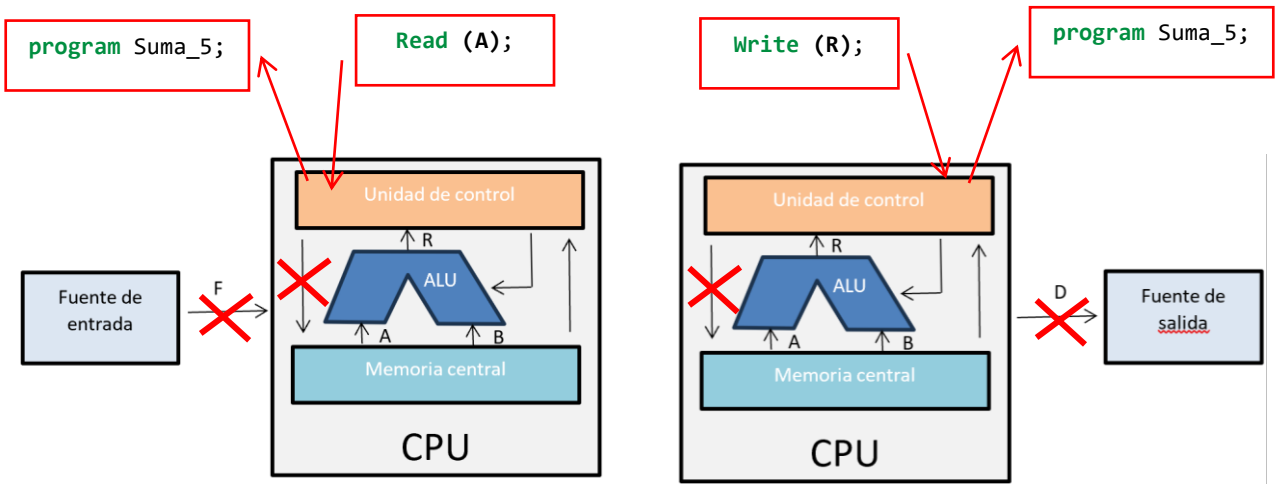




Tras el título y la especificación, que es más bien información semántica y descriptiva para el desarrollador, respectivamente; **la unidad de control recibe la instrucción donde se declaran** tres **variables**, A, B y R de tipo entero. Estas variables no son más que **huecos de memoria vacíos que se reservan para el almacenamiento** de un entero. Así, **la unidad de control envía una señal a la memoria central para que esta ejecute dicha acción**.

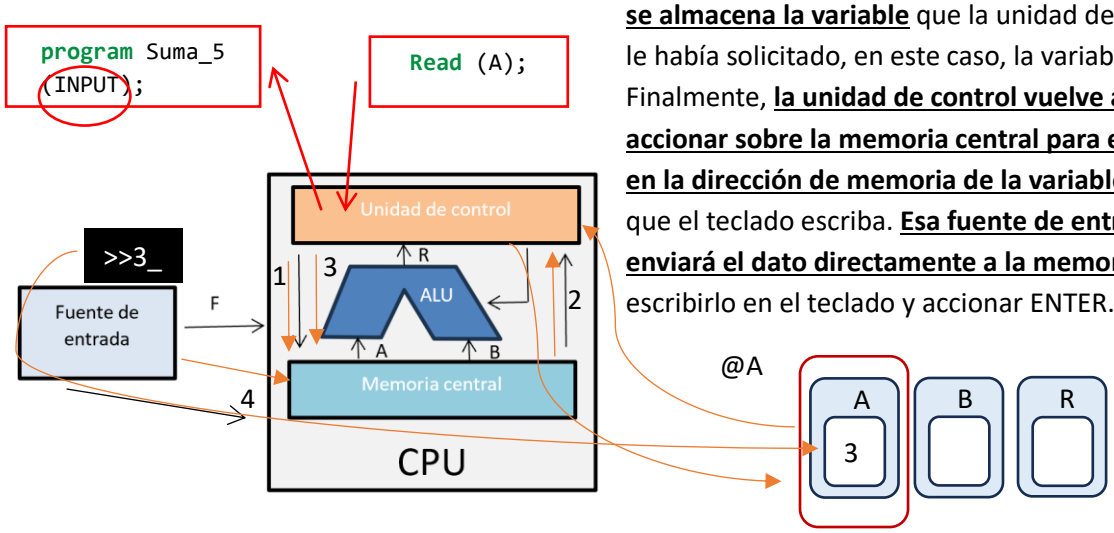
A continuación, se procede a ejecutar el algoritmo del programa. En este caso, debido a la sintaxis del lenguaje, la unidad de control lo procesa porque recibe la palabra reservada `begin`, la cual indica el inicio del algoritmo.

En primer lugar, tenemos la instrucción `Read (A)`, que lee el entero por el teclado y lo almacena en la variable A. Importante saber bien estas dos cosas relacionadas con la E/S. En primer lugar, **la unidad de control no enviará una señal a la memoria central** (que es la que permite almacenar el dato en la variable) **si no se ha habilitado previamente la fuente de entrada o de salida** (en este caso, mediante la etiqueta INPUT u OUTPUT), resultando en un **error de compilación**.



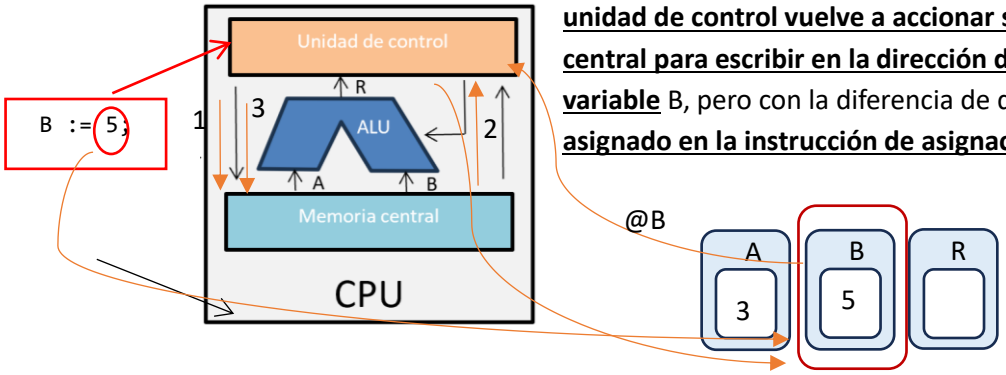
Lo segundo, **al introducir un dato por el teclado, hay que asegurarse de que lo que se introduzca pertenece al tipo de dato que se solicita**. Así mismo, si se pide un entero no se pueden introducir ni números decimales, ni letras, ni oraciones ni nada distinto a un número entero. Si no se hiciera, provocaría un **error de compilación** por la no correspondencia de lo que se recibe y lo que se desea almacenar.

Prosiguiendo con la instrucción de **lectura de datos, si la etiqueta INPUT está habilitada**, la unidad de control llamará a la memoria central y le advertirá de que va a recibir un dato por el teclado, y por tanto **el bus de entrada de datos se habilitará**. Posteriormente, **la memoria manda a la unidad de control la dirección de memoria donde se almacena la variable** que la unidad de control le había solicitado, en este caso, la variable A. Finalmente, **la unidad de control vuelve a accionar sobre la memoria central para escribir en la dirección de memoria de la variable** A lo que el teclado escriba. **Esa fuente de entrada, enviará el dato directamente a la memoria** tras escribirlo en el teclado y accionar ENTER.



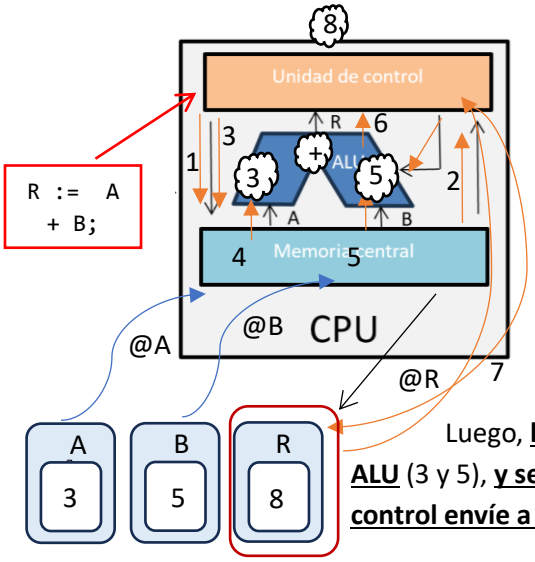
Pasamos a la siguiente instrucción `B := 5`, bastante similar, pero **almacena el dato** directamente dada una instrucción que asigna el valor 5 a la variable B, **por acción del código en sí** y no por fuentes externas al procesador.

Los pasos que se ejecutan son exactamente iguales salvo el último, en el que **la unidad de control vuelve a accionar sobre la memoria central para escribir en la dirección de memoria de la variable** B, pero con la diferencia de que **escribirá lo asignado en la instrucción de asignación (:=)**.



A continuación, ejecutamos la siguiente instrucción `R := A + B`. En esta línea de código, se deben sumar los valores almacenados en las variables A y B y almacenar dicho resultado en la variable R.

Para ello se hará uso de la **ALU**, del inglés, **Unidad Aritmético-Lógica**. La ALU es un componente esencial para el **proceso de datos** y la ejecución correcta de las operaciones sobre los datos.



Como de costumbre, en primer lugar, **la unidad de control lanza una señal a la memoria central** y se le advierte de la operación que se desea ejecutar.

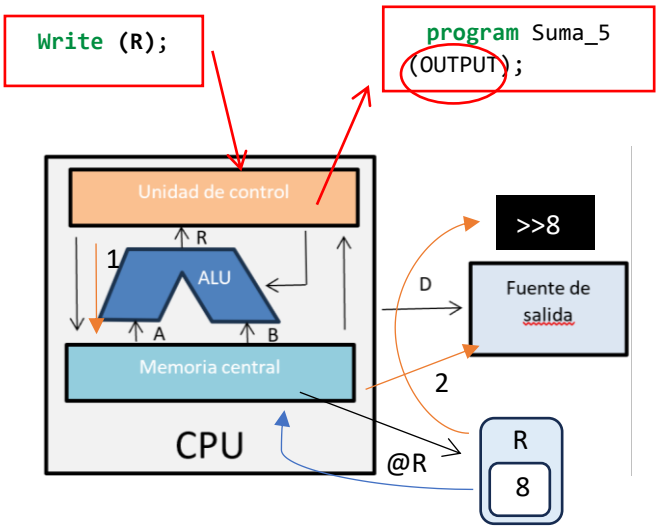
La memoria envía a la unidad de control la dirección de memoria de la variable de destino (R) **y busca las direcciones de memoria de las variables fuente** (A y B), es decir, los operandos.

Luego, **la memoria envía los valores de los operandos a la ALU** (3 y 5), **y se operarán según el operando que la unidad de control envíe a la ALU** (+)

Finalmente, **la ALU opera la operación** (3 + 5) **y emite un resultado** (8). **Este resultado se escribirá en la dirección de memoria de la variable destino** (R) una vez la unidad de control ejerza una señal sobre la memoria.

Y en el paso final, ejecutaremos la instrucción `Write (R)`, la cual **escribirá el dato** mostrándolo en la pantalla.

Si la etiqueta OUTPUT está habilitada, la unidad de control llamará a la memoria central y le advertirá de que uno de los datos que tiene guardados va a imprimirse en pantalla, y por tanto, **se habilitará el bus de salida de datos.**



La memoria buscará la dirección donde se almacenó la variable solicitada por la unidad de control (R) **e imprimirá su valor en pantalla** (8).

Esta breve explicación del funcionamiento del procesador nos da una breve idea de lo que se realiza en la caja negra, pero no podemos excedernos de confianza, ya que sólo estamos viendo su superficie. Sin embargo, nos da una idea muy aproximada a la hora de ejecutar un programa del porqué de los errores que podemos tener en nuestro código, ¡y habrá para un rato!

Y es que cuando programamos, muchas veces pensamos como un ser humano, cuando lo suyo sería pensar como un ordenador, por ello es importante saber un poco acerca del procesador, el cerebro del ordenador, puesto la lógica que a nuestro parecer parecería algo muy obvio, el ordenador necesitaría otra manera de comprender el mismo concepto, nuestros cerebros no están hechos del mismo material.

1. Ideas del discurso: Algoritmos

Es difícil encontrar una definición exacta a lo que se denomina algoritmo, puesto en muchos libros de texto aparece erróneamente como un sinónimo de programa. Según la RAE, un algoritmo es un **conjunto ordenado y finito de operaciones que permite hallar la solución de un problema**. Esta definición se confunde con la de programa, que según también la RAE, se define como un **conjunto de instrucciones basadas en un lenguaje de programación** específico que un ordenador usa para **resolver un problema determinado**. (véase Tema 0)

De hecho además, en la definición de algoritmo que nos ofrece la RAE encontramos los primeros fallos. Lo único cierto en sí de la definición es el **orden**, no podemos seguir los pasos al tuntún, no se puede primero meter un huevo en la sartén si primero no sacamos la sartén, ni tampoco si no echamos el aceite.

Un algoritmo no tiene por qué ser finito, por ejemplo, la serie de pasos para obtener todos los números primos que existen es infinita, puesto existen infinitos números primos. Sin embargo, **la capacidad de la computadora sí que es finita**, no cabe el infinito en el disco duro, por lo que si se ejecutara el **algoritmo teórico** en el ordenador, **acabaría hasta corromperse y desbordar** al punto en el que no quepa ninguna cifra más. Es por ello que se deben limitar estos algoritmos de infinitos pasos.

Por último, **un algoritmo no tiene por qué resolver un problema en específico, aunque siempre son base de los programas**. Esto puede variar según la ejecución del programa. Por ejemplo, en un programa de consulta telefónica se realiza un algoritmo en el que según se pulse a cada tecla numérica del 0 al 9, se realice una tarea u otra. P.ej: para pedir cita, pulse 1; para anular cita, pulse 2; para consultar cita, pulse 3; para finalizar la consulta, pulse 0. Tras pulsar la tecla, se ejecutarán los algoritmos correspondientes a la operación que se debe realizar tras elegir una opción determinada.

Esto último dicho traza la fina línea roja que distingue un programa de un algoritmo. **Un algoritmo es una lista de instrucciones para efectuar paso por paso un proceso determinado**. Y así está el quiz de la cuestión, el distinguir entre el **problema o propósito** (que es genérico) y el **proceso o rompecabezas** (que es específico). Por ejemplo, un gestor de venta de entradas para cine, teatro o estadio cumple con el propósito de vender las entradas, pero debe pasar por ciertos procesos como la selección de sesión, la selección de asientos, la selección de aperitivos y finalmente la de pago. En cada proceso se debe implementar un proceso o algoritmo de selección.

En este curso, los programas que diseñaremos serán bastante sencillos en comparación con las grandes aplicaciones funcionales que nos podemos encontrar en el mercado digital, puesto nuestro objetivo es la introducción al mundo de la programación. Por ello, lo más común a la hora de programar, es que solamente tengamos que diseñar un algoritmo por cada programa. En conclusión, **el programa es la implementación del algoritmo en un lenguaje de programación concreto**. Ya habíamos dicho anteriormente en el tema 0 que para entablar cualquier tipo de discurso que queramos, solamente se necesitaría un medio y un idioma. **Ese medio será el ordenador y se empleará el lenguaje de programación a nuestro antojo**. **El tema de conversación o idea del discurso será** la serie de pasos que se deben ejecutar para desempeñar el proceso, es decir, **el algoritmo**, como se había dicho antes, parte nuclear del programa.

Lo que se va a estudiar a continuación no será por ahora la implementación del algoritmo en el programa, sino el desarrollo del algoritmo fuera del programa.

Según la mayoría de convenios de la Informática, **un algoritmo debe cumplir las siguientes características**:

1. ***Tiempo discretizado y finito***. El algoritmo debe **ejecutarse paso a paso**, esto significa que cada paso del algoritmo consume la misma unidad de tiempo; mediante una **serie finita de pasos** y por tanto su tiempo de ejecución siempre será finito. O lo que es lo mismo, **el algoritmo debe terminar**.
2. ***Descripción secuencial de estados***. El algoritmo debe por cada paso que se dé, **a partir de un estado computacional inicial determinado, obtener el estado final tras la ejecución del paso**. Se denomina estado de un programa a la configuración única de información que almacena la computadora en ese instante determinado como tras una línea de código. El **primer estado** inicial antes de ejecutar el algoritmo completo se denomina **precondición** y el **último estado** que se obtiene después del último paso, es decir, cuando ya se finaliza el programa, es la **postcondición**. Las **pre-post condiciones obtenidas conforman la especificación del algoritmo**, una descripción breve de los datos que deben entrar (input) y los datos que deben salir (output) para su correcta ejecución. Entraremos en dicho detalle en el Tema 2.
3. ***Exploración acotada***. La transición de estados queda completamente determinada por una **descripción discreta y finita**; es decir, entre cada estado y el siguiente solamente existe una variación en una cantidad fija y limitada de términos del estado actual. Por ejemplo, tras la instrucción `x := x + 1` el estado inicial antes de ejecutar dicha asignación solamente variará la variable x una unidad con respecto al estado final.

En resumen, un algoritmo es cualquier cosa que pueda **ejecutarse paso a paso**, donde cada paso se pueda **describir sin ambigüedad** (se detallará esto más adelante) y **en cualquier medio posible**, y además **tiene un límite fijo en cuanto a la cantidad de datos que se pueden transmitir en un solo paso**. Sin darnos cuenta, ejecutamos diariamente algoritmos. P.ej: una receta de cocina, una coreografía, una operación aritmética o una reservación de una línea aérea.

La ejecutabilidad de un algoritmo es muy flexible, puesto no es en realidad un programa, sino una guía o plan para realizar el proceso. Luego, **cualquier medio y cualquier lengua es posible** de emplear a la hora de diseñar un algoritmo. Entre los medios destacan **el lenguaje natural, el diagrama de flujo, el pseudocódigo y los lenguajes de programación**.

P.ej. La receta del huevo frito de nuestro manual de cocina, es un algoritmo:

<<1. Tomamos la sartén y la ponemos en la vitro a fuego medio
2. Echamos un poco de aceite de oliva
3. Sacamos el huevo de la nevera y lo cascamos
4. Metemos el huevo en la sartén y lo dejamos 5 minutos
5. Con una espátula, sacamos el huevo frito y lo dejamos en un plato
6. Acompaña el plato con una hogaza de pan ¡Que aproveche!>>

Pero jamás seremos capaces de ver este algoritmo expresado en lenguaje natural implementado en el programa `freir_huevo`, ya que… ¿qué funcionalidad tendría? No todos los

algoritmos se pueden programar, a no ser que recurramos a la **abstracción**. ¿Qué pasa si quisiéramos emplear este algoritmo para un juego de cocina?

La **abstracción en programación** se refiere al énfasis en el **"¿qué hace?"** más que en el "¿cómo lo hace?" Por una parte, en el programa ya hemos visto (en el tema 0) qué hace (seguir las indicaciones de un texto en un lenguaje de programación determinado) y cómo lo hace (asumiendo que nuestro procesador tenía arquitectura de Von Neumann) Por otra parte, **en el algoritmo no nos interesa cómo lo hace el usuario**, volviendo otra vez a la **caja negra** donde entran datos y salen datos. Como ejemplo pondremos una coreografía. Esta se puede hacer con lenguaje corporal (bailando), en una lista de instrucciones para una guía de baile o incluso diseñar un programa para programarla en un jugador de un videojuego. Haciéndolo de la manera que queramos, la coreografía no dejará de ser un algoritmo, ya que interpretan claramente series de pasos, series de quehaceres.

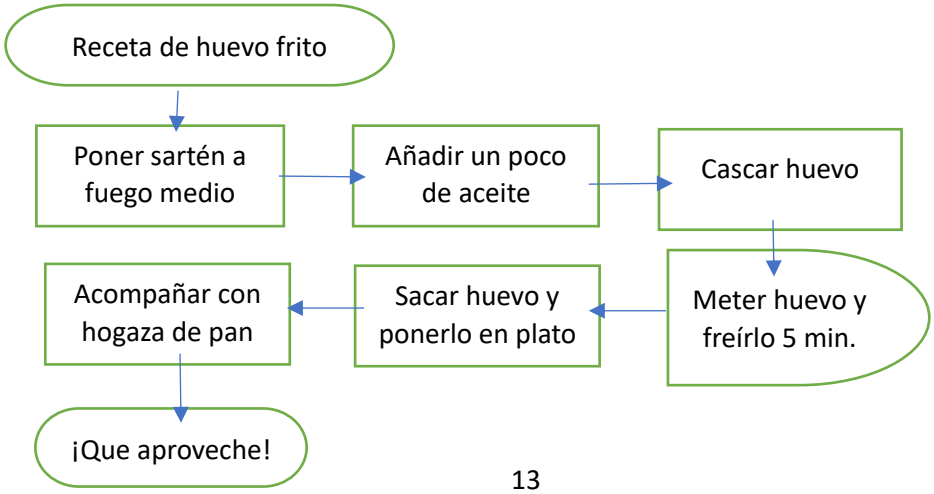
La importancia de la abstracción radica en **evitar la ambigüedad que se pueda cometer usando el lenguaje natural**, por ello, no es buena idea diseñar algoritmos con nuestras propias palabras. Por ejemplo, la frase “he visto a una persona con unos prismáticos” puede tener dos interpretaciones distintas.

Cuando de un enunciado se infiere más de una interpretación posible, decimos que el enunciado es ambiguo. Luego, si este enunciado formara parte de un algoritmo, los desarrolladores se podrían fácilmente decantar por una de las dos interpretaciones y no cumplir con su cometido.

Por ello, se requieren formas más abstractas para expresar los algoritmos.

1.1 Diagramas de flujo

El diagrama de flujo es un **esquema que se utiliza para representar un algoritmo**, el cual **ilustra las operaciones a efectuar y en qué secuencia se ejecutarán**. Su finalidad es esquematizar un proceso. Por ejemplo, la receta del huevo frito se podría esquematizar de la siguiente manera:

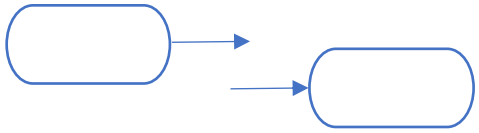
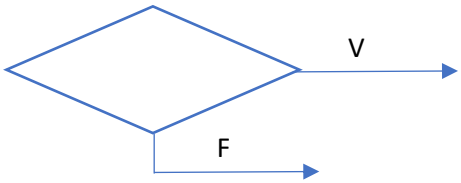


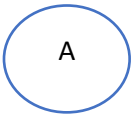
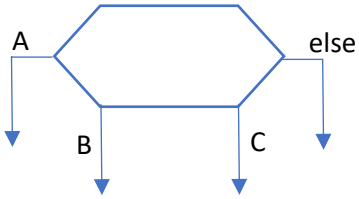
Ambas formas de expresión de la receta simbolizan el mismo algoritmo. Además, este tipo de gráficos no sólo se utilizan en el contexto de **la programación**, sino que también tiene otras utilidades en otras disciplinas como **la economía, la tecnología industrial y la psicología**.

Como has podido observar, el **lenguaje natural** de estos diagramas destaca por ser **más simple pero comprensible** para el usuario, suelen ser enunciados con un verbo en infinitivo que indican un **imperativo**, sugieren una orden directa que manda realizar **la ejecución del paso**; aunque en otros campos distintos de la programación también nos encontramos sustantivos que denotan **nombres de subprocesos** pertenecientes a un **proceso más complicado** como la venta de un producto o la fabricación de un material.

Esto dicho anteriormente marca la diferencia entre **dos tipos de diagramas de flujo**: el **diagrama analítico**, que detalla paso por paso la ejecución de un proceso; y el **diagrama sinóptico**, que resume la ejecución de un proceso complejo a través de subprocesos más sencillos y comprensibles.

A continuación, se presentarán los siguientes símbolos o **bloques del diagrama de flujo** adoptados por el estándar ISO 5807 de la Organización Internacional de Normalización que hoy día sigue vigente:

	***Líneas de flujo***. Muestran el orden en el que se operan los pasos.
	***Bloque de iniciación o terminación***. Indica el principio o el final del algoritmo. Todo diagrama debe comenzar y finalizar con este bloque.
	***Bloque de asignación***. Representa una asignación de un valor o bien numérico o bien de una operación aritmética a una variable.
	***Bloque de decisión***. Dependiendo de la veracidad del enunciado, ejecutará uno de los dos caminos disponibles. Es una pregunta del tipo Sí/No. Sus líneas de flujo estarán etiquetadas mediante las letras V o F.
	***Bloque de entrada de datos***. Indica las variables que necesitan ser introducidas por el teclado.
	***Bloque de salida de datos***. Indica las variables que se imprimirán en pantalla.
	***Bloque de comentario***. Se usa para hacer anotaciones en el diagrama, la línea señala el

	bloque sobre el que se quiere comentar, y el bloque se coloca aparte de la secuencia.
	***Bloque de proceso predefinido***. Se usará a partir del Tema 9 para indicar Subprogramas. Se explicará en su respectivo tema.
	***Conectores***. Cuando las líneas de flujo son largas o confusas, se utilizan dos pares de conectores etiquetados con letras y sustituyen a la línea.
	Sin embargo, cuando el par de ***conectores*** sitúa a cada uno de ellos ***en distintas páginas***, adquieren otra forma.
	***Bloque de demora***. Indica el tiempo que se debe parar la ejecución del algoritmo tras la ejecución de dicho paso.
	***Bloque de condición***. Dependiendo del valor que tenga la variable o expresión y según las condición que cumpla, etiquetada en cada línea, ejecutará un camino u otro. Si no está etiquetada, se entenderá como si tuviera de etiqueta "else", es decir, cuando no cumple ninguna de las anteriores condiciones.

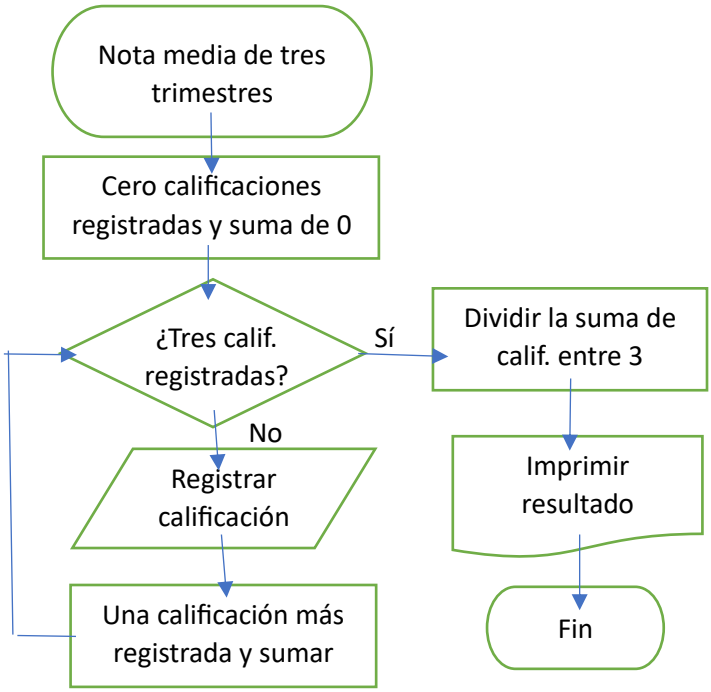
Estos bloques son válidos para cualquier diagrama de flujo que se realice indiferentemente de su funcionalidad y del país en el que se desarrolle, **son esquemas utilizados a nivel internacional** y aprobados por el consenso de todos los estados miembros de dicha organización.

Existen también dos convenios más a tener en cuenta a la hora de diseñar los diagramas de flujo. En primer lugar**, la dirección del flujo por convenio debe desarrollarse de arriba a abajo o de izquierda a derecha, excepto en el caso de que** no quepan en la página y **se pretenda enroscar la secuencia** (véase el diagrama de flujo del huevo frito), siendo el primer pliegue el que siga la dirección por convenio y evitando obstaculizar el normal desarrollo de los otros elementos del diagrama. En segundo lugar, sobre el diseño de las flechas que direccionan el flujo del diagrama, **estas flechas siempre serán poligonales y sus correspondientes lados tendrán dirección vertical u horizontal**. Los estilos L o corchete [para las flechas de flujo, por ejemplo, cumplimentan con su normalización.

La ventaja de utilizar el diagrama de flujo como medio de expresión de un algoritmo es que **facilita la comprensión del algoritmo**, debido a que **el cerebro humano reconoce muy fácilmente las formas**; siendo más fácil identificar errores e ideas principales. Sin embargo, tiene la desventaja de que **su diseño no se parece al del programa**, y traducir el diagrama a un

pseudocódigo legible requiere un paso extra. Además, **cuanto más grande sea el problema, más complejo será el diagrama**, y por tanto, más difícil de seguir.

Como ejemplo, se expone a continuación a la derecha mediante un diagrama de flujo un algoritmo que calcula la nota media de los exámenes de tres trimestres e imprime el resultado en pantalla. Para ello, se pide al usuario que primero introduzca la nota de primer trimestre, luego la del segundo y finalmente la del tercero, en una secuencia. Por ejemplo "7.5 8 9.2"



Si lo tradujéramos a lenguaje natural sería bastante más complejo:

<<¿Cómo calcular la nota media de tres trimestres?

1. Iniciamos el contador de calificaciones registradas a cero, así como su suma total
2. ¿Se han registrado las tres calificaciones?
3. En caso negativo, solicitar al cliente que introduzca por el teclado una calificación
4. Contar una calificación más registrada y sumarla
5. Volver al paso 2
6. En caso afirmativo, dividir la suma de calificaciones entre tres
7. Imprimir dicho resultado>>

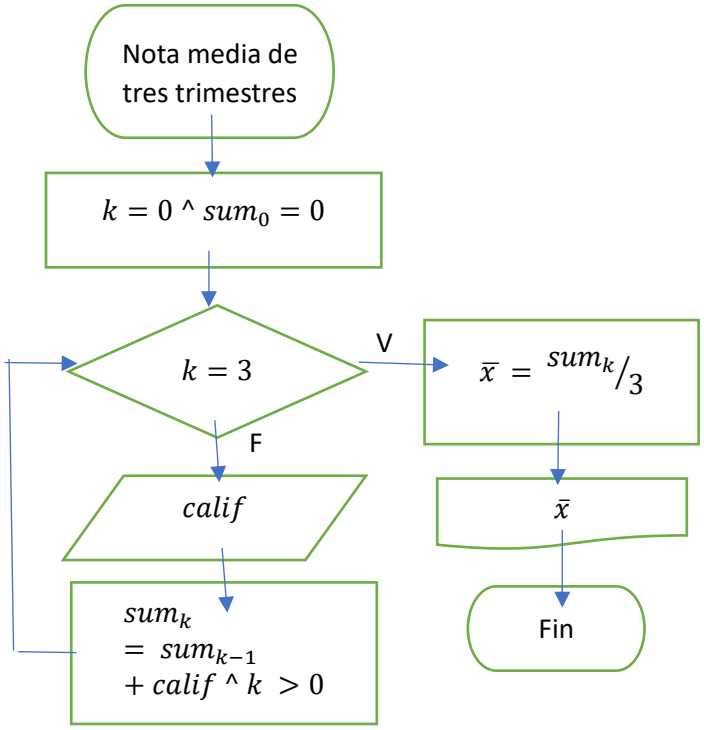
Se puede observar que **las formas guían fácilmente hacia la comprensión del algoritmo**, identificando lo que hace cada bloque y guiando a través de sus flechas el flujo del algoritmo; y que **el lenguaje de diagrama de flujo es más sencillo**, con enunciados cortos con un **verbo en infinitivo** cuando se trata de una **orden** o un **sustantivo** que define el **estado de computación** en el que se halla en ese momento el algoritmo. Además, en **lenguaje natural**, aparte de ser **más desarrollado**, sucede lo que se denomina **salto de línea**, por lo que en cada decisión que hay que tomar o acción en bucle nos obliga a saltar y dirigirnos a las líneas que deseemos.

Es más, este diagrama todavía puede mejorar si recurrimos a un **lenguaje abstracto** como el **algebraico** o el **lenguaje formal de diagramas de flujo**. El lenguaje algebraico ya lo conocemos, es el que permite que en un problema como por ejemplo "La suma de edades de los tres hijos es 30. Las edades de los dos menores sumadas resultan la edad del mayor. El mayor tiene el triple de años que el menor" permita convertir las cantidades desconocidas en incógnitas y las proporciones en coeficientes.

Así, llamando x a la edad del mayor, y, a la del mediano y z, a la del menor, obtenemos los enunciados "x + y + z = 30; x = y + z; x = 3z" Esto confiere abstracción a las edades, y **trata**

estas **como incógnitas de otra identidad abstracta manejable** como un sistema de ecuaciones del que no nos importa si las incógnitas son edades, son frutas o son lo que nos convenga en ese momento.

Sin embargo, el lenguaje algebraico cuenta con **expresiones más complejas** como fracciones, potencias, raíces cuadradas, símbolos especiales, etc. que **el ordenador** no es capaz de leerlos, ya que tienen una **capacidad limitada de reconocer dichas estructuras** de la misma forma que nosotros mismos al escribir por el teclado no podríamos escribir símbolos como π, √ , o ^ , debemos utilizar un editor de documentos que los introduzca.



O incluso existen **enunciados** de lenguaje natural como en el algoritmo anterior **que deben ser traducidos como funciones**, ya que "a la suma anterior le sumamos la nueva calificación" sólo se puede interpretar como $sum_k = sum_{k-1} + calif$ cuando $k > 0$, asumiendo que sum_k es un término de una sucesión sum_n de k términos, siendo $sum_0 = 0$.

Si expresáramos el algoritmo anterior con **lenguaje algebraico**, podría verse de la siguiente forma aquí arriba. Sin embargo, aquí encontramos la siguiente pega. ¿El algoritmo indica expresamente que sum_k es un término de una sucesión o que hay que hallar el siguiente término de la sucesión y que, por lo tanto, hay que incrementar en una unidad k? Está claro que este tipo de lenguaje **se comprende sólo por gente con estudios matemáticos avanzados**, y que hay que comprender también el concepto de sucesión para comprender el algoritmo.

Buscaremos **un lenguaje más sencillo que sea formal, que sea abstracto, y que sea más fácil de comprender**, no sólo para cualquier párvulo que haga la cuenta de la vieja en su puesto de dependiente, sino también **para el ordenador**, como si fuera una especie de código que se le da machacado para que lo comprenda el ordenador. Esto es el **lenguaje formal de diagramas de flujo**

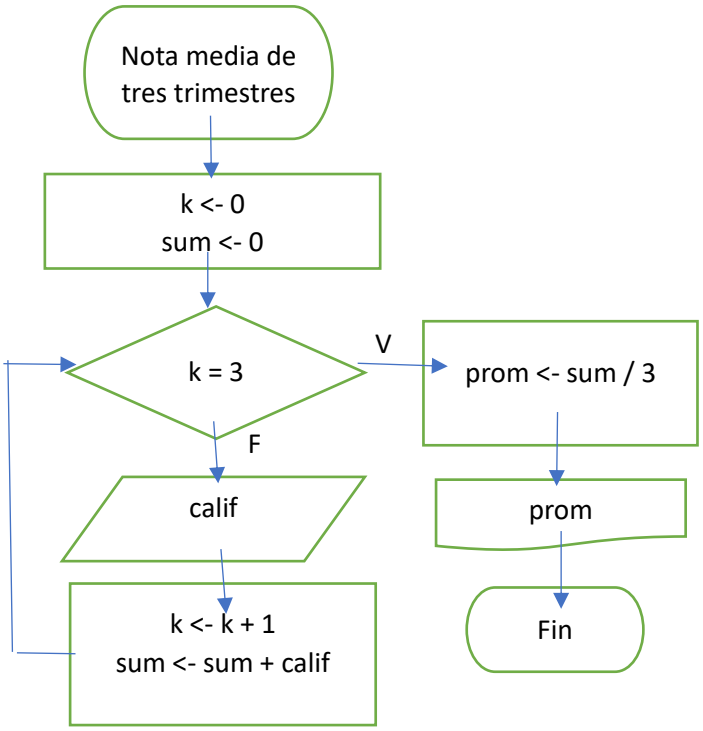
Este lenguaje **destaca por su parecido a los primeros lenguajes de programación creados, por su simplicidad y su linealidad**. Sus reglas son las siguientes:

- Los nombres de las **variables** se representan con **letras** (A, B, x, y), **abreviaturas** (dist, area, vol) o **combinaciones cualquiera de letras y números**.
- Como **toda representación de letras y números denota una variable**, expresiones matemáticas comunes como 4ac denotarían variables. Si se deseara representarlo como producto (*), se denotaría 4*a*c, con el **signo de producto expreso**.

- **La denotación de símbolos especiales** como √x, |x| o sin x **se realizará mediante funciones matemáticas**, en su caso, sqrt(x), abs(x) y sin(x)
- **Las potencias y las fracciones** como a^b o $\frac{a}{b}$ **deben escribirse como a^b y a/b**, respectivamente.
- **El orden de las operaciones** a ejecutar en operaciones matemáticas **se mantiene** según los mismos convenios matemáticos, siendo este paréntesis, funciones, potencias, productos, fracciones, la negación, la suma y la resta.
- **Las operaciones lógicas se denotarán por el nombre de la puerta lógica que designan**. Así, escribiremos not, and, or, xor en lugar de ¬, ^, ˅, ↔.
- **Las instrucciones de asignación se denotan con el formato <<variable <- valorUoperacion>>**, ahora se verá mejor con el nuevo diagrama de flujo.
- **Los operadores relacionales**, usados sobre todo en los bloques de decisión, se expresan como =, <> (o /=), >, >=, <, <=.

Así p.ej., $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$ se escribiría como x_1 <- (-b+sqrt(b^2-4*a*c))/(2*a).

El diagrama de flujo redactado en su lenguaje formal resultaría de la siguiente forma:



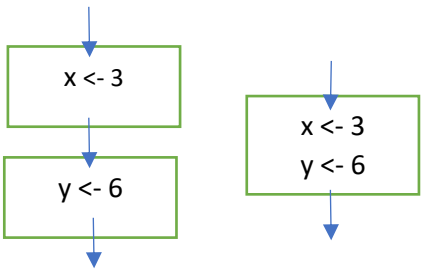
Este lenguaje formal nos expone la principal **diferencia entre la igualdad y la asignación** en el contexto de la programación, y que su concepto no es el matemático.

Cuando en matemáticas designamos a una incógnita un valor, **en programación asignamos un valor a una variable**, es decir, reservamos **un hueco de memoria con el nombre de esa variable y le ponemos valor**. Por otra parte**, la igualdad designa una expresión booleana que se evaluará como verdadero o falso**, no es una ecuación ni una expresión algebraica.

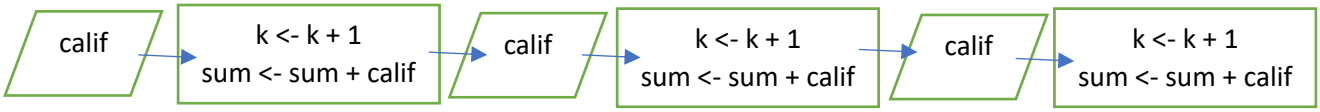
Esto permite expresiones como k <- k + 1, es decir, "asigno a la variable k el valor que tiene actualmente k más uno" Hay que tener en cuenta también que una expresión algebraica como x = x + 1 no tiene solución, por lo que por reducción al absurdo no tiene sentido hablar de igualdad en las variables, sino de asignación.

Dos cosas sobre el diagrama de flujo se deberían destacar ahora basadas en el anterior ejemplo.

En primer lugar, **cuando hay más de una asignación secuencialmente, estas se pueden acumular en el mismo bloque de asignación**.



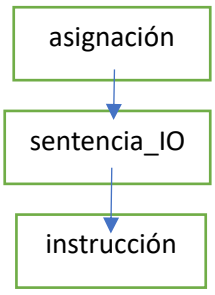
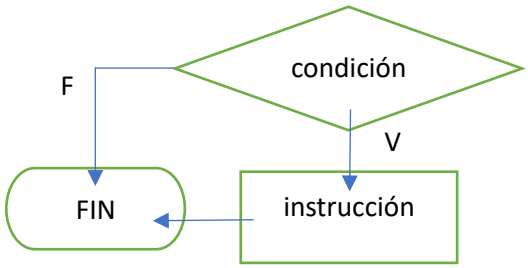
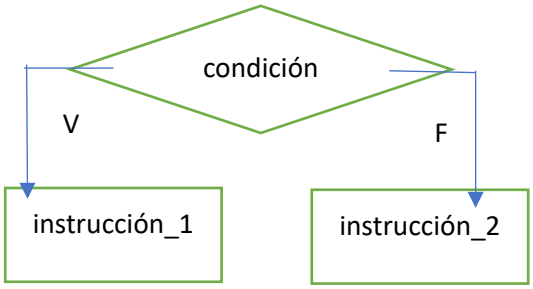
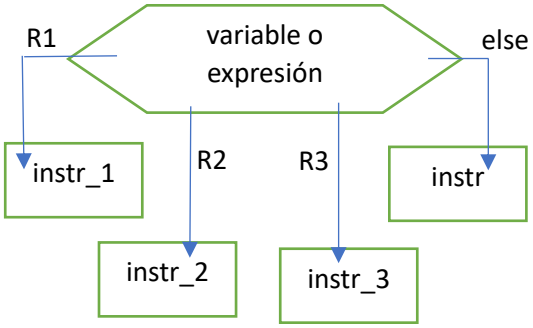
En segundo lugar, **el bucle**, que destaca por su flecha estilo [corchete, **sustituye a una sucesión de repetidas asignaciones**, ahorrando bastante espacio. En el ejemplo anterior, el bucle podría sustituirse ineficientemente por:



1.2 Pseudocódigo

Otra de las formas para expresar algoritmos consiste en emplear el **pseudocódigo**. Este consiste en un **lenguaje utilizado para escribir los algoritmos** de los programas de la computadora en una especie de **lenguaje natural** bastante **simplificado pero** bastante **adaptado a los lenguajes de programación**, lo que permite transcribirlos más fácilmente a código.

Para entender mejor lo que es el pseudocódigo, veremos algunas de las **estructuras de control** típicas que podemos encontrarnos al **graficar un diagrama de flujo**. Estas estructuras corresponden con una forma peculiar de codificación o lenguaje informal llamado pseudocódigo.

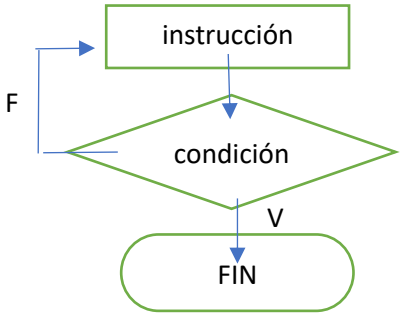
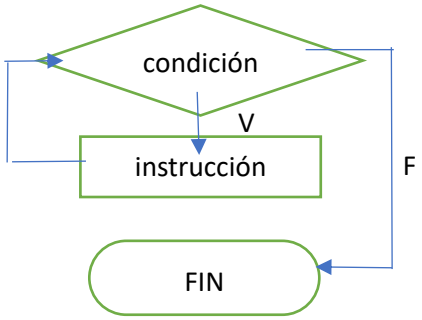
<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td><code><asignación>::= <variable> := <valor> |
<expresión> ;
<sentencia_IO>::= scan() | print() ;
<instrucción>::= <asignación> |
<sentencia_IO>;</code></td>
<td><code><condición>::= <expresión_booleana>;
<bloque_if>::=
if (<condición>) then
    <instrucción>; {Se cumple la cond.}</code></td>
</tr>
<tr>
<td><u>Estructura secuencial</u>.
Es aquella que ejecuta una sucesión de instrucciones secuencialmente, ejecutándolas todas y sólo una vez.</td>
<td><u>Estructura selectiva simple</u>.
Es aquella que se compone de un bloque de decisión en el cual, si y solo si la condición se cumple, se ejecuta una determinada instrucción.</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
</table>

<table>
<tr>
<td>

```
<bloque_if_else>::=
if (<condición>) then
    <instrucción_1>; {Se cumple la cond}
else
    <instrucción_2>; {No se cumple}
```

</td>
<td>

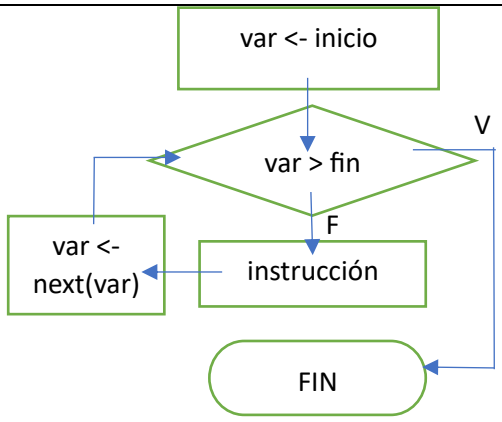
```
<bloque_cases>::=
cases of <variable> | <expresión>
     <rango_1>: <instrucción_1>;
     <rango_2>: <instrucción_2>;
     <rango_3>: <instrucción_3>;
     else: <instrucción>; {otros rangos}
```

</td>
</tr>
<tr>
<td><u>Estructura selectiva doble</u>.
Es aquella que se compone de un bloque de decisión en el cual, si la condición se cumple, se ejecuta una determinada instrucción. En caso contrario, ejecuta una instrucción alternativa.</td>
<td><u>Estructura selectiva múltiple</u>.
Se compone de un bloque de condición, en el cual, dependiendo de la condición que cumpla la expresión, ejecutará una instrucción u otra. Si esta es “else”, se ejecutará en caso de que no cumpla ninguna de las condiciones anteriores.</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>

```
<bloque_repeat>::=
repeat
     <instrucción>;
until (<condición>);
```

</td>
<td>

```
<bloque_while>::=
while (<condición>) do
    <instrucción>;
```

</td>
</tr>
<tr>
<td><u>Estructura iterativa “REPEAT”</u>.
Es aquella que contiene un bucle y, tras la ejecución de una acción, vuelve a repetirla hasta que se cumpla una condición de salida.</td>
<td><u>Estructura iterativa “WHILE”</u>.
Es aquella que contiene un bucle y ejecuta una acción mientras la condición de esta siga cumpliéndose.</td>
</tr>
<tr>
<td rowspan="2"> 
</td>
<td>

```
<bloque_for>::=
for <var> in range <inicio> to <fin> do
    <instrucción>;
```

</td>
</tr>
<tr>
<td><u>Estructura iterativa “FOR”</u>.
Es aquella que contiene un bucle y repite una acción en función de un rango de valores discretizado y finito de una variable. La función next() expresada en el diagrama devuelve el valor siguiente correspondiente según el valor de la variable y el tipo de dato. P.ej. del 0 le siguen 1, 2, 3… Pero de A le siguen B, C, D…</td>
</tr>
</table>

Evidentemente, **<u>al ser el pseudocódigo un lenguaje natural, existen otras formas de expresar estas estructuras</u>**. Para ello, nos podemos basar incluso en otro lenguaje de programación que nos resulte familiar.

Por ejemplo, en lugar de usar las palabras “then” o “do” para las condicionales o bucles respectivamente, se pueden usar llaves o bráquets {}, cerrando el conjunto de instrucciones perteneciente a dichos bloques, e incluso se podría colocar indicativos como “begin” o “end” que indican el inicio y el fin del bloque. Ejemplos:

```
if (<condición>)then
    <instrucción>;
```

```
if (<condición>) {
    <instrucción>;
}
```

```
if (<condición>)then
begin
        <instrucción>;
end if;
```

Sin embargo, existe un convenio conforme al diseño de algoritmos relacionado con la **sangría del código**, y se debería cumplirse, pues facilita la **legibilidad del algoritmo**. Este convenio indica que **a las instrucciones que pertenezcan a un bloque determinado se les debe** aplicar un carácter tabulador delante de la instrucción, prefijándola y que marque **más sangría con respecto al resto de líneas de código**.

En caso contrario, la lectura del código se vería ofuscada. Sí es cierto que **expresiones lineales para los bloques** del tipo `if (<condición>) then <instrucción>;` **se pueden encontrar incluso en lenguajes de programación**, poniéndolas al mismo margen que las asignaciones o las sentencias.

El problema sucede cuando existe más de una instrucción perteneciente al bloque y/o se indexan al menos dos bloques, como por ejemplo un bloque if dentro de un bucle. P.ej.

```
i:=2; while (i<=size) do if (v[i-1]>v[i]) then aux:=v[i-1]; v[i-1]:=v[i]; v[i]:=aux; i:=i+1;
```

Como se puede observar, esta línea de código **tiende a la ambigüedad**, **y eso no es un algoritmo**, pues incumple la característica tres: exploración acotada, puesto ya no existe una descripción discreta y finita, sino que puede haber varias interpretaciones al respecto, porque no se han acotado los bloques. Estas serían dos posibles interpretaciones, corrigiendo este algoritmo al aplicarle sangría:

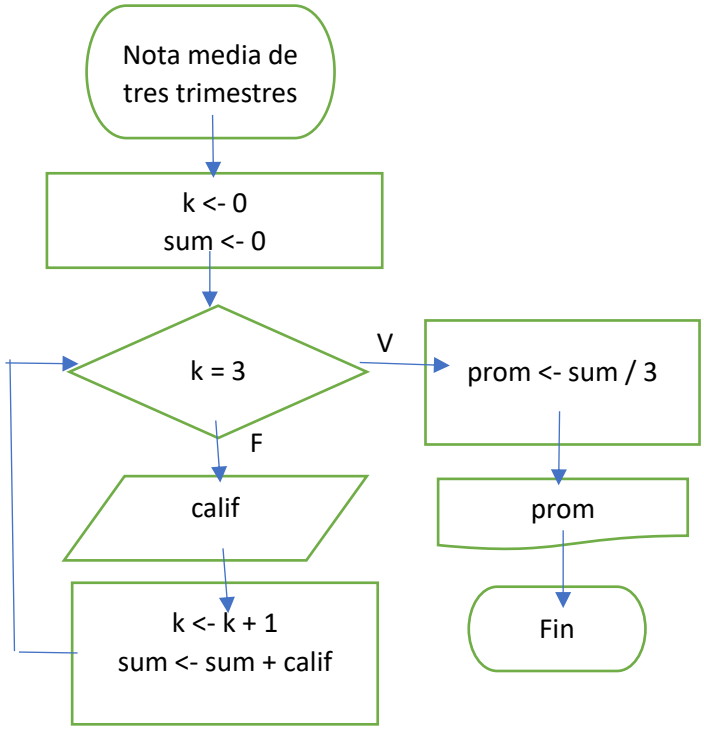
```
i:=2;
while (i<=size) do
    if (v[i-1]>v[i]) then
        aux:=v[i-1];
    end if;
end while;
v[i-1]:=v[i];
v[i]:=aux;
i:=i+1;
```

```
i:=2;
while (i<=size) do
    if (v[i-1]>v[i]) then
        aux:=v[i-1];
        v[i-1]:=v[i];
        v[i]:=aux;
    end if;
    i:=i+1;
end while;
```

En realidad, la interpretación correcta es la segunda, la de la derecha, y se puede intuir leyendo atentamente el programa entero, pero para uno darse cuenta de ese detalle hay que tener bastante experiencia en la programación, lo que evidentemente, al llamarse este curso Bases de la Programación, no tenemos. Se han añadido adicionalmente los indicadores de fin de bloque (ends) para que se vea mejor gráficamente que según la interpretación, se le considera a ciertas instrucciones como parte de dichos bloques o no.

Y por ello se estableció dicho convenio, al cual se le añade la regla de **escribir cada instrucción en una línea distinta**, como si fuera una lista de instrucciones a ejecutar en una receta, evitando expresiones como `instr_1; instr_2; instr_3;` y expresando dicha secuencia como se muestra a la derecha. Aún así, se podría hacer como la primera manera si y sólo si las instrucciones son todas asignaciones o todas sentencias de entrada o salida.

```
instr_1;
instr_2;
instr_3;
```




```
mediaTresTrimestres(calif; prom){
sum:=0;
for k in range 0 to 2 do
      scan(calif);
      sum:=sum+calif;
prom:=sum/3;
print(prom);
}
```

```
mediaTresTrimestres(calif; prom){
k:=0; sum:=0;
while (k<3) do
      scan(calif);
      sum:=sum+calif;
      k:=k+1;
prom:=sum/3;
print(prom);
}
```

Se puede observar el diagrama de flujo correspondiente al algoritmo anterior que calculaba la media de tres trimestres. A la derecha, tenemos dos posibles implementaciones en pseudocódigo, una siguiendo la estructura de un bloque for, y otra, la del bucle while. Pero, ¿no ha habido aquí dos interpretaciones distintas del mismo algoritmo? No, no es como el anterior caso, en el que había dos interpretaciones distintas de dos posibles algoritmos porque la acotación de los bloques no era clara.

Estos dos fragmentos de código distintos en los que se delimitan claramente los bloques pero que **resuelven el mismo problema** son dos algoritmos distintos que llevan a distintos caminos pero al mismo destino. Recuerda que todos los caminos llevan a Roma.

Y no, **no incumpliría para nada la regla de exploración acotada, porque la serie de pasos a seguir describe cantidades discretas y finitas** en ambos casos, la única diferencia es que un bloque aumenta el valor k por sí solo, el bloque for; y el otro hay que incrementárselo añadiendo una asignación de más, el bloque while.

Si la descripción de estados de dos algoritmos distintos es equivalente, podemos decir que ambos algoritmos resuelven el mismo problema. Verificaremos este ejemplo más adelante en el tema 2.

1.3 Lenguajes y tipos

Finalmente, hay que dar el gran paso adelante, que es **implementar uno** de los dos **algoritmos en un programa** de verdad, **con un lenguaje de programación** que nuestro gólem de hierro sepa.

Se define **lenguaje de programación** como el **conjunto de reglas gramaticales que instruyen a que un ordenador se comporte de una manera determinada**. Es un **lenguaje formal y artificial**, creado por el ser humano para su comprensión, evitando confusiones y errores, pero también es legible por la computadora. Los ordenadores son muy precisos en

cuanto a las instrucciones que se reciben, que son cerradas, y las ejecutan gracias a dos herramientas clave, **el compilador y el interpretador**.

La diferencia entre estas dos herramientas es que el interpretador lee una por una las instrucciones y las traduce una por una al lenguaje máquina, el único que entiende el ordenador; mientras que el compilador lee todo el programa y lo traduce entero y de seguida.

Estas herramientas **transforman el lenguaje de programación en un lenguaje comprensible por la computadora, este es el lenguaje máquina**, un sistema de códigos directamente interpretable por un microcircuito programable. Es específico de la arquitectura o diseño físico del ordenador.

Por ejemplo, Intel y AMD tienen sus propios diseños de ordenador, por lo que tendrán dos lenguajes máquina ligeramente distintos. Sin embargo, **el hecho de que estos diseñadores de computadores tengan un lenguaje máquina común a todos los procesadores de su empresa evita una Torre de Babel** en la que un programador tendría que aprender un nuevo lenguaje de programación para cada ordenador distinto.

Ahora bien, si el lenguaje máquina es universal para todos los procesadores del mismo diseño, ¿por qué no se aprende este y ya está? Pues porque **es bastante más fácil aprender un lenguaje de programación con sentencias comprensibles** por el ser humano que un lenguaje máquina de 0's y 1's.

Los lenguajes de programación se clasifican según varios criterios:

- ***Clasificación por generaciones***: se hace una clasificación según la etapa histórica o **generación** a la que pertenece el lenguaje, abarcan desde **lenguajes máquinas** (1ª), **de bajo nivel** (2ª), los **de alto nivel** que vamos a estudiar (3ª), hasta nuevos lenguajes en los que se utilizan **herramientas prefabricadas** como Scratch o App Inventor (4ª) o incluso **inteligencia artificial** (5ª)
- ***Lenguajes de alto o de bajo nivel***: cuanto **mayor es el nivel del lenguaje, mayor es su nivel de abstracción**, es decir, menos se parece al lenguaje máquina pero más se parece al lenguaje natural. Nosotros veremos lenguajes de alto nivel.
- ***Clasificación por propósito***: existen lenguajes como MatLab (MATrix LABoratory) o XML (bases de datos) que sólo cumplen con una serie de problemas en específico. **Los que son capaces de resolver cualquier problema**, que son los que utilizaremos, se denominan **lenguajes de propósito general**
- ***Tipado fuerte o débil***: Veremos ambos tipos de lenguajes. Los de **tipado fuerte** deben **declarar el tipo de la variable antes de ejecutar el algoritmo** y **no permite** utilizar para esa variable **un tipo de dato distinto** a no ser que se convierta en el tipo estipulado (como por ejemplo Ada o C), mientras que los de **tipado débil no controlan los tipos de datos de las variables** que se utilizan, siendo posible utilizar variables de cualquier tipo de dato en un mismo escenario (como Python o JavaScript)

Existen muchas más clasificaciones de lenguajes, pero me he resumido a las más importantes para no complicar el tema.

A continuación, se mostrará un ejemplo de cómo se implementaría en un programa uno de los dos algoritmos que resolvían el problema anterior. Hay que recordar que **se necesita cumplir con la estructura del programa**, es decir, su título, especificación, declaración de

variables y finalmente, el algoritmo. Yo por ejemplo elegiré el algoritmo con el bucle for y lo programaré en varios lenguajes, elegiré Pascal, Ada, Python y C.

```pascal
(* PASCAL *)
program mediaTresTrimestres (INPUT,
OUTPUT);
var
	k: Integer;
	sum, calif, prom: Real;
begin
	sum:=0.0;
	for k:= 1 to 3 do
	begin
		Read(calif);
		sum:=sum+calif;
	end;
	prom:=sum/3.0;
	Write(prom);
end.
```

```ada
-- ADA --
with Ada.Float_Text_IO; use
Ada.Float_Text_IO;
procedure mediaTresTrimestres is
	k: Integer;
	sum, calif, prom: Float;
begin
	sum:=0.0;
	for k in 1..3 loop
		Get(calif);
		sum:=sum+calif;
	end loop;
	prom:=sum/3.0;
	Put(prom);
end mediaTresTrimestres;
```

```python
# PYTHON - mediaTresTrimestres #
sum = 0.0
for k in range (3):
	calif = input()
	sum+=calif
prom=sum/3.0
print(prom)
```

```c
/* C */
#include <stdio.h>
void main(){
	int k;
	float sum, calif, prom;
	sum=0.0;
	for (k=0; k<3; k++){
		scanf("%f", calif);
		sum+=calif;
	}
	prom=sum/3.0;
	printf("%f", prom);
}
```

En este caso, la especificación del programa es lo único que no hemos escrito en los programas para ahorrar espacio y observar con mayor claridad las diferencias entre los distintos lenguajes de programación, especificaremos el programa en el siguiente tema. El criterio que he utilizado para señalar las distintas partes del programa es el siguiente:

- ***Verde y negrita***: son **palabras reservadas**, es decir, palabras que tienen **significado gramatical** en ese lenguaje y se constituyen en una estructura de control determinada, por lo que **no sirven para declarar variables**. Por ejemplo, en ADA no se pueden llamar a las variables "loop", "begin" o "is" Cuando es verde pero no es negrita, indican instrucciones de entrada/salida
- ***Azul turquesa***: corresponden a **comentarios** o anotaciones
- ***Marrón claro***: líneas de código que permiten la **habilitación de E/S**
- ***Naranja oscuro***: **nombre del tipo** de dato (entero, real, booleano, carácter, cadena de caracteres, etc.)
- ***Gris claro***: **valor del tipo** de dato
- ***Negro***: **nombres** (de programa, de variable, etc.), **símbolos ortográficos** (p.ej. punto y coma al final de la instrucción, llaves para delimitar bloques, etc.) y **otros**

El ejercicio de **programar** resulta en realidad en un ejercicio de **traducir y cumplir con las reglas gramaticales que nos ofrece el lenguaje desde un programa escrito en pseudocódigo**.

Podemos recalcar perfectamente **dos notables diferencias entre tipado fuerte y tipado débil**. En **tipado fuerte**, el programa está **estructurado**, de tal forma que se incluye el título del programa y la **declaración previa de variables y de su tipo de dato** a utilizar, además de la **habilitación previa de sus herramientas de E/S**, cosas que en **tipado débil** (y en el pseudocódigo) no ocurre, **su código es bastante más libre**, sin delimitar y con un título puesto a modo de comentario (excepto en el pseudocódigo, que tiene cabecera y en él se indican las variables de entrada y salida), se declaran variables en cualquier parte del código y no es necesario ni indicar el tipo de dato que se desea introducir ni solicitar la habilitación de la E/S, estos se detectan por sí solos.

Entonces, ¿cuál es la idea de programar? ¿Cómo empezaríamos a dar nuestros primeros pasos? Pues bien, primero de todo, habría que entender el enunciado, y comprender lo que se pide, obviamente. En segundo lugar, deberíamos organizarnos las ideas, escribiendo en una lista la secuencia de pasos a ejecutar, intentando adaptarnos al **diagrama de flujo**. Es altamente recomendable adaptarnos cuanto antes a diseñar diagramas de flujo y a utilizar su **lenguaje formal**, por lo menos durante nuestros primeros programas, ya que si podemos organizarnos fácilmente las ideas con este esquema gráfico, entonces el tercer paso, que es escribir en **pseudocódigo**, será pan comido, ya que sería buscar la **estructura** correspondiente, la más parecida, y traducir desde esa base.

Finalmente, desde el pseudocódigo, al tener un gran parecido al código, es bastante más fácil traducirlo al **lenguaje de programación** deseado. Para ello, se recurriría a una **guía rápida para usar el lenguaje** y buscamos la **regla sintáctica formal** escrita en un sistema llamado **Expresión-S** de la sentencia o estructura que deseamos codificar. Además, yo he adjuntado algunos ejemplos válidos para tener una referencia de la regla sintáctica y comprenderla mejor. Usaremos los lenguajes de programación a partir del tema 4.

1.4. Ejercicios propuestos

Se sugieren realizar los siguientes ejercicios para comprobar que se han comprendido los conceptos más importantes de este tema. Aunque estos ejercicios no sirvan para programar como tal, sí que forman parte de unos conocimientos previos que hay que tener dominados antes de empezar a programar de verdad. Estos ejercicios se deben realizar usando diagramas de flujo o pseudocódigo. No es estrictamente necesario, aunque sí recomendado, usar la sintaxis de este libro, puedes usar tus propias palabras.

1. (*) Diseña un diagrama de flujo en el que se calcule la expresión $y = \frac{1}{4}x + \frac{4}{5}$ (desarrolla los diagramas de flujo en lenguaje formal)
2. (*) Diseña un diagrama de flujo en el que se calcule la expresión $y = \frac{\frac{1}{3}\left(x^3 - \frac{1}{2}x + \frac{1}{25}\right)^2}{30} - 25$
3. (*) Diseña un diagrama de flujo en el que, dados el número de lados de un polígono regular, la longitud del lado y su apotema, calcule el área del polígono. Recuerde que el área de un polígono regular es la mitad del producto de su perímetro y su apotema
4. (*) Diseña un diagrama de flujo en el que se lean dos números X e Y, se calcule X elevado a Y y se imprima el resultado
5. (*) Diseña un diagrama de flujo en el que se pida al usuario el radio del círculo, se calcule su área y se imprima el resultado. Recuerde que π es igual a 3.14159265... y que la fórmula del área es $A = \pi r^2$

6. (*) Diseña un diagrama de flujo en el que dados tres coeficientes por el usuario a, b, c; resuelva la ecuación cuadrática $ax^2 + bx + c = 0$ e imprima sus soluciones de la forma <<x_1=valor1, x_2=valor2>> Ten en cuenta que las cadenas de caracteres deben escribirse entre comillas, para diferenciarse de las variables. P.ej. "x_1=", valor1… Recuerda que la solución de esta ecuación es $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$
7. (*) Diseña un diagrama de flujo en el que dado por el usuario un valor en euros, se conviertan euros a pesetas, libras, dólares y yuanes en ese orden; y se muestre la equivalencia en una tabla donde se indique de cada uno el nombre de la ocurrencia y su valor. Aquí abajo se muestra un ejemplo.

Euros	1
Pesetas	166
Libras	0.8
Dólares	1.1
Yuanes	7.9

8. (**) Diseña un diagrama de flujo en el que se da un número y se determine si es par. Para ello se debe utilizar la op. matemática a mod b, representada por el operador (%), que devuelve el resto de la división producida entre a, b
9. (**) Diseña un diagrama de flujo en el que dados tres coeficientes a, b, c; se verifique que formen una terna pitagórica (cumplen su teorema, $a^2 + b^2 = c^2$)
10. (**) Diseña un diagrama de flujo en el que se lea un número por el teclado y se indique si este es positivo, negativo o cero en la pantalla
11. (**) Diseña un diagrama de flujo en el que dada por el usuario una nota de un control, se califique como SUSPENSO (<5), APROBADO (5-7), NOTABLE (7-9) o SOBRESALIENTE (>9) y se muestre en pantalla
12. (**) Diseña un diagrama de flujo en el que se impriman los 10 primeros números impares.
13. (**) Diseña un diagrama de flujo en el que se impriman todos los múltiplos de 3 y de 5 hasta cierto supremo dado por el usuario
14. (**) Diseña un diagrama de flujo en el que se lea un número natural y se devuelva en pantalla su descomposición. P.ej. para 1234 sería 4*1 + 3*10 + 2*100 + 1*1000
15. (***) Diseña un diagrama de flujo en el que se introduzca por el teclado una secuencia de números y se muestre en pantalla el mayor de todos. Utiliza eoln() como expresión booleana que indica si se ha llegado al final de la línea o no en la lectura de la secuencia
16. (***) Diseña un diagrama de flujo en el que se introduzca por el teclado una secuencia de números naturales y se muestre en pantalla, cuando se lea el valor -1, la suma de todos ellos.
17. (***) Diseña un diagrama de flujo en el que se introduzca por el teclado una secuencia de números y se muestre en pantalla la media aritmética de todos ellos
18. (***) Diseña un diagrama de flujo en el que se introduzca por el teclado una secuencia de números y se reimprima la misma secuencia pero sin números negativos
19. (***) Escribe en pseudocódigo el diagrama de flujo correspondiente al ejercicio 6
20. (***) Escribe en pseudocódigo el diagrama de flujo correspondiente al ejercicio 7
21. (***) Escribe en pseudocódigo el diagrama de flujo correspondiente al ejercicio 11
22. (***) Escribe en pseudocódigo el diagrama de flujo correspondiente al ejercicio 12
23. (***) Escribe en pseudocódigo el diagrama de flujo correspondiente al ejercicio 13
24. (***) Escribe en pseudocódigo el diagrama de flujo correspondiente al ejercicio 14
25. (****) Escribe en pseudocódigo el diagrama de flujo correspondiente al ejercicio 16

26. (****) Diseña un diagrama de flujo en el que se impriman las tablas de multiplicar
27. (****) Diseña un diagrama de flujo en el que dado un número cualquiera, guarde el número de dígitos pares que contiene y la longitud de la secuencia más larga
28. (****) Diseña un diagrama de flujo en el que dada la altura de un triángulo por el teclado, imprima una escalera descendente de asteriscos. Por ejemplo, aquí se muestra la salida por pantalla esperada para h=2, h=3 y h=4, respectivamente. Para dar los saltos de línea, utiliza el carácter "\n"

```
*          *          *
**         **         **
           ***        ***
                      ****
```

2. Descripción del discurso: Diseño por contrato

Como bien hemos dicho anteriormente, **el uso del lenguaje natural tiende a la ambigüedad**, pues muchas veces se basa en el contexto. Como ejemplo habíamos descrito anteriormente la acción de "ver a una persona con unos prismáticos" ¿La persona que los portaba era el observador o el observado?

A la hora de describir el programa que se debe realizar, **ocurrirá lo mismo si nuestra descripción no es la adecuada**, si nuestra descripción es ambigua, si **existe más de una interpretación posible** a un mismo enunciado.

Por ejemplo, si nos solicitaran un programa que calculara el seno de x al cuadrado, existen dos posibles interpretaciones: $sin(x^2)$ o $sin^2 x$. **Necesitamos concretar nuestro lenguaje natural, describiendo con más precisión el procedimiento que debemos realizar**. Respectivamente, podrían describirse dichas opciones como "el seno de la función x al cuadrado" y "el cuadrado de la función seno de x"

Esta **descripción precisa del comportamiento esperado de un programa** se denomina **especificación**, y establece una importante **relación entre los datos de entrada permitidos y los datos de salida esperados**.

En este caso, se asume que x es un número real, pero ¿la medida del ángulo viene dada en grados o en radianes? No se ha descrito el programa adecuadamente, pues el enunciado sigue siendo ambiguo. Para evitar la ambigüedad, una especificación ideal debería contener una **descripción adecuada tanto de los datos de entrada**, p.ej. "Un número real x que expresa la medida de un ángulo en radianes" **como de los de salida**, p.ej "El cuadrado de la función seno de x" Siguiendo este ejemplo, la especificación de nuestro programa podría perfectamente ser: "El programa calcula el cuadrado de la función seno de x, siendo x un real que expresa la medida de un ángulo en radianes"

Cuando se nos pida diseñar un programa, en algunos casos se nos planteará un **enunciado** el cual es **ambiguo**, como en el caso anterior. Una adecuada especificación deberá servir para eliminar cualquier ambigüedad que pueda desprenderse del enunciado.

Si no se nos da la especificación o no es adecuada, entonces se corre el riesgo de que el problema que se nos pida no sea el adecuado, por lo que entonces **podríamos recurrir a cualquiera de las posibles interpretaciones**, especificando cuál de ella se ha tomado.

Ahora bien, **si el contexto del enunciado es suficiente** como para inferir cuál es la interpretación más adecuada, entonces **nos basaremos en el contexto para diseñar nuestro programa**. En cuanto vayamos avanzando en este libro, nos encontraremos enunciados mucho más extensos, del tamaño de una página como mínimo, por lo que en este tipo de problemas radicará el contexto.

La especificación describe, en resumen, **qué es lo que hace el programa**, y no cómo. Para una misma especificación, existen muchos caminos diferentes para llegar a la misma conclusión, es decir, **cualquier algoritmo que se ajuste a las limitaciones de los datos de entrada permitidos y los datos de salida esperados, puede implementarse correctamente en un programa con dicha especificación**.

Esto que parece una tontería, ya que el propio código del programa describe qué hace el programa y cómo lo hace, tiene gran importancia. **Una buena especificación hace comprensible el objetivo del programa** y por lo tanto, **sirve como documentación** de este. Esta también **ahorra costes de mantenimiento** al **facilitar la reutilización** del algoritmo.

Además, **facilita la verificación del programa** al tener instrucciones claras de qué hace el programa, qué valores recibe y qué valores emite, como si fuera una **función matemática** donde entran cosas y salen cosas, volviendo al símil de la **caja negra**. Si dado un **certificado o caso de prueba**, al introducir el valor de entrada en un programa, obtenemos el de salida esperado, **el programa es parcialmente correcto**. Si se certifican **todos los casos de prueba**, el programa es **totalmente correcto**.

Se podría decir que una **buena especificación** es aquella **descripción clara** (fácil de entender), **breve** (sin redundancias) **y precisa** (sin ambigüedad) **del quehacer de un programa**, además de tener un notable **equilibrio entre la concisión y la generalidad** (sin detalles superfluos pero sin omisión de detalles cruciales).

La especificación ha de ser entendida como un contrato. Tomemos como ejemplo cualquier electrodoméstico. Cuando compramos un producto siempre viene acompañado de un manual de instrucciones. En este manual se nos detalla qué se considera un uso correcto del aparato y las cláusulas de garantía.

En algunos casos, la especificación nos servirá para **restringir los casos posibles**. Esto puede suceder cuando, si no existiese ninguna condición para el dato de entrada, el **problema** que se nos plantea sería **muy costoso**. En otros casos ocurre que al establecer restricciones se **facilita el algoritmo** y al mismo no afectaría al uso normal del programa porque **se sabe que el usuario nunca va a realizar tal acción**. Por ejemplo, introducir una edad negativa.

Cuando se diseña un producto **hay que tener en cuenta todos los casos de uso que se pueden presentar**; por ejemplo, alguien podría decidir conectar el aparato a una red de 300V cuando se indica explícitamente en el manual que no se puede conectar a corrientes superiores a 250V. **Las especificaciones** (en este caso, los modos de uso) **deben restringir los usos no previstos o estudiados** del aparato, deben indicar bajo qué condiciones se espera que sea utilizado. Así, si estas condiciones quedan claramente

especificadas y es el usuario el que no las sigue, entonces la responsabilidad es del usuario, quedando el fabricante eximido de toda responsabilidad si el producto no funciona.

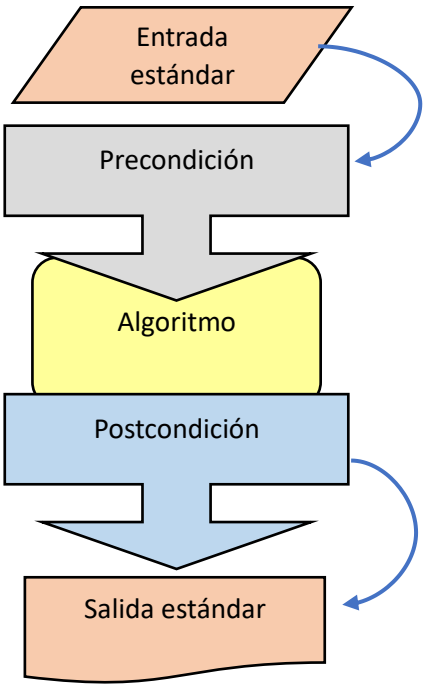
¿Cuál es el objetivo de la especificación en programación? Aplicar la misma idea, diseñar por contrato un programa.

El **diseño por contrato** fue diseñado por Bertrand Meyer, y lo aplicó a su propio lenguaje de programación, Eiffel. Este diseño abarca también ciertas metodologías y formalismos en la programación los cuales omitiremos, al ser este un curso de Bases de la Programación, resumiendo este diseño a lo más importante. A este diseño simplificado lo llamaremos **especificación pre-post**.

2.1 Especificación pre-post

Se denomina así porque esta especificación **sólo da la descripción de los datos de entrada permitidos y la de los de salida esperados**, aunque suficiente para diseñar un programa. La especificación se compone principalmente de seis partes:

- **Datos de entrada**: tipos de dato de los inputs y sus nombres
- **Entrada estándar**: nombre de los datos que se reciben por el teclado
- **Precondición**: condición de los datos de entrada que debe satisfacerse antes de la ejecución del algoritmo
- **Datos de salida**: tipos de dato de los outputs y sus nombres
- **Salida estándar**: nombre de los datos que se imprimen en pantalla
- **Postcondición**: condición de los datos de salida que debe satisfacerse justamente tras la ejecución del algoritmo



Por ejemplo, la especificación pre-post del ejemplo del capítulo anterior, del algoritmo que calculaba la media de tres trimestres podría ser:

- **Datos de entrada**: una secuencia de reales “calif”
- **Entrada estándar**: sus reales “calif”
- **Precondición**: 0.0 <= calif <= 10.0 y secuencia de longitud 3
- **Datos de salida**: un real “prom”
- **Salida estándar**: el real “prom”
- **Postcondición**: prom = la media de las tres calificaciones, 0.0 <= prom <= 10.0

Otra especificación en lenguaje natural no ambigua de este programa podría ser:

“Este programa lee por el teclado una secuencia de tres números reales comprendidos entre 0 y 10, y corresponden a la nota de cada trimestre que un alumno ha obtenido en la asignatura, e imprime en pantalla un número real que indica la media aritmética de las tres notas”

Por otra parte, en enunciados ambiguos como en el que se nos solicitaba calcular el seno de x al cuadrado, cualquiera de estas interpretaciones podrían ser especificaciones válidas:

a) Interpretación 1: el seno de la función x al cuadrado, x se mide en grados
   - **Datos de entrada**: un real "x"
   - **Entrada estándar**: no solicitada
   - **Precondición**: x está medido en grados, 0 <= x < 360
   - **Datos de salida**: un real "result"
   - **Salida estándar**: no solicitada
   - **Postcondición**: result = seno(x^2), -1 <= result <= 1

b) Interpretación 2: el seno de la función x al cuadrado, x se mide en radianes
   - **Datos de entrada**: un real "x"
   - **Entrada estándar**: no solicitada
   - **Precondición**: x está medido en radianes, 0 <= x < 2π
   - **Datos de salida**: un real "result"
   - **Salida estándar**: no solicitada
   - **Postcondición**: result = seno(x^2), -1 <= result <= 1

c) Interpretación 3: el cuadrado de la función seno de x, x se mide en grados
   - **Datos de entrada**: un real "x"
   - **Entrada estándar**: no solicitada
   - **Precondición**: x está medido en grados, 0 <= x < 360
   - **Datos de salida**: un real "result"
   - **Salida estándar**: no solicitada
   - **Postcondición**: result = (seno x)^2, result <= 1

d) Interpretación 4: el cuadrado de la función seno de x, x se mide en radianes
   - **Datos de entrada**: un real "x"
   - **Entrada estándar**: no solicitada
   - **Precondición**: x está medido en radianes, 0 <= x < 2π
   - **Datos de salida**: un real "result"
   - **Salida estándar**: no solicitada
   - **Postcondición**: result = (seno x)^2, result <= 1

Por lo tanto, si en este enunciado ambiguo no se da la especificación, a la hora de diseñar el programa, somos libres de elegir cualquiera de estas cuatro opciones siempre y cuando el contexto no sea suficiente como para determinar cuál es la interpretación más adecuada, indicaremos cuál es la descripción o especificación que se ha utilizado.

Uno de los tres conceptos que debemos tener en cuenta a la hora de especificar pre-post un programa es, la diferenciación clara entre tipo de dato y condición del dato.

Es muy importante saber diferenciar entre el tipo de dato y la condición que debe cumplir el dato. Un **tipo de dato** puede ser por ejemplo un booleano, un entero, un String, un vector de nº enteros, etc. algo que caracterice a los datos que se vayan a usar en el algoritmo y que formen parte de un **conjunto ya definido**, se detallará bastante más su definición en el siguiente tema; mientras que la **condición del dato** es una **característica muy específica** del dato **en un programa determinado** y que no se puede exactamente clasificar en un conjunto definido.

Por ejemplo, para un algoritmo que divide dos números X, Y cualquiera, se debería expresar en la precondición del programa {y /= 0}, como una característica específica de Y en la división de dos números sean naturales, enteros, reales, complejos, etc., siendo esto último dicho el tipo de dato que puede ser X, Y.

Si no fuera necesaria una precondición para el dato de entrada y se restringe solamente al tipo de dato que es este, se podría omitir la pre o decir más formalmente que la pre es {true}

También se debe tener en cuenta que **el tipo de dato que se utilice podría variar la descripción del algoritmo original el cual implementa otro tipo determinado**. Por ejemplo, la especificación del algoritmo del producto de dos números X e Y utilizando su definición matemática = > $x * y = x + x + x \dots ,\ y\ veces,\ x, y \in \mathbb{N}$ cuando estos son naturales en contraposición con cuando estos son enteros, o bien manteniendo el algoritmo anterior o bien modificándolo para abarcar todos los posibles casos que dicho tipo de dato nos pueda ofrecer.

	Algoritmo original / Tipo de dato A	**Algoritmo original / Tipo de dato B**	**Algoritmo adaptado / Tipo de dato B**
Input	Dos naturales, X e Y	Dos enteros, X e Y	Dos enteros, X e Y
Output	Un natural Z	Un entero Z	Un entero Z
Pre	true	x,y >= 0	true
Post	Si x or y = 0, z=0 Si x and y /= 0, z = x + x + x… y veces	Si x or y = 0, z=0 Si x and y /= 0, z = x + x + x… y veces	Si x or y = 0, z=0 Si (x>0 and y>0) or (x<0 and y<0) , z = |x|+|x|+|x|… |y| veces Si (x>0 and y<0) or (x<0 and y>0) , z = -|x|-|x|-|x|… |y| veces
Result	Espec. original	Alteración de la pre	Alteración de la post

Observamos la especificación del programa con el tipo de dato A y la comparamos con la especificación del nuevo programa que usa el algoritmo original pero cambia el tipo de dato que se utiliza a B. Ahora X, Y son enteros, pero con la especificación en tipo A, el tipo de dato utilizado B es incompatible con la afirmación de la postcondición de A, puesto no existe el sumatorio cuando su nº de términos (y) a calcular de la sucesión (x) es negativo, es decir, no puedes sumar un número cualquiera -3 veces, carece de sentido. Por ello, para mantener el algoritmo original, debemos indicar qué tipos de enteros queremos, restringirlos a una característica específica de ellos, y por ello, tanto x como y deben ser mayores o iguales que 0, aunque eso es sinónimo de ser naturales, limitando los enteros a un conjunto ya definido y por lo tanto, se podría considerar este como tipo de dato. Luego, ¿es correcto precondicionar los enteros X, Y? ¿Es correcto definir los naturales como un tipo de dato?

No entraremos en debate ahora en lo que puede o no definirse como tipo de dato, pero sí podemos afirmar que cualquier natural es un entero, pero no todos los enteros, como son los negativos, son naturales. Al considerar el tipo de dato utilizado como naturales, no existe ninguna característica específica que limite cuáles naturales hay que utilizar y cuáles no, pues todos se pueden utilizar y el algoritmo sigue siendo correcto. Esto no sucede en el caso de que se consideren enteros, pues existe una característica específica que limita cuáles enteros hay que utilizar y cuáles no, y es que como hemos citado antes, no podemos usar negativos, por lo que X, Y deben ser mayores o iguales que cero, dando una precondición de lo que debe cumplir el entero para que el algoritmo sea correcto.

Para que este algoritmo de producto de enteros sea más eficiente, debemos usar todos los números enteros existentes, añadiendo por supuestísimo los negativos y llegar al objetivo de poder multiplicar cualquier par de nº enteros sin ninguna precondición. Pero para ello, hemos tenido que modificar la postcondición de la especificación, y por tanto, adaptado el algoritmo a todos los posibles casos con el nuevo tipo de dato.

Una vez aclarada esta común confusión entre tipo de dato y condición del dato, pasemos a los otros dos puntos importantes a la hora de especificar pre-post.

El siguiente concepto se refiere al **nivel de restricción de las especificaciones**. Si las afirmaciones correspondientes a la pre o a la post son **más restrictivas**, se denominan **condiciones fuertes**. Sin embargo, cuando son más permisivas, se denomina **condiciones débiles**. P.ej. ser un número primo es una condición más fuerte que ser un número par.

Una de las principales tareas del programador a la hora de especificar un programa es convertir conjuntos de condiciones débiles en fuertes. **Cuanto más fuerte sea una condición, más seguro será el contrato**. Por ejemplo, estas conjunciones de condiciones débiles implican estas condiciones fuertes:

- x>=0 and x/=0 => x>0
- (x or y) and not(x and y) => x xor y
- N divisible entre 1 y N, no lo es para todos los nº entre 2 y N-1 => N es un nº primo

Como casos extremos, la afirmación {True} es la condición más débil posible mientras que la afirmación {False} es la condición más fuerte posible.

Otra característica más sobre las restricciones de condiciones a destacar es que **las restricciones de las variables son propagables**, de tal forma que si un término está acotado entre dos valores, cualquier variable compuesta por una combinación lineal de variables acotadas también lo será.

Por ejemplo, volviendo al problema de la media, si 0 <= calif <= 10, entonces la suma sum = calif1 + calif2 + calif3 se acota en 0+0+0 <= sum <= 10+10+10, 0 <= sum <= 30, y el promedio prom = sum/3, se acota en 0/3 <= prom <= 30/3, 0 <= prom <= 10, tal y como se indica en la especificación.

Finalmente, con respecto al **uso de las herramientas de E/S, se debe indicar su uso explícitamente** en la especificación del algoritmo. Si no se indica, se asume que los datos de entrada y de salida se leen y escriben en memoria. Se debe indicar correspondientemente que se recibe por **entrada estándar** o se **lee por el teclado** y que se emite por **salida estándar** o se **imprime en la pantalla**, respectivamente.

2.2 Casos de prueba

Para finalizar este bloque, en donde hemos sido capaces de diseñar algoritmos muy sencillos y describirlos, debemos ser capaces de argumentar por qué nuestro algoritmo es válido, por qué es correcto. La **corrección** es una propiedad intrínseca del algoritmo, y se puede definir como la **coincidencia entre el comportamiento real del programa y el del programa pretendido** o bien como la **coincidencia entre lo que se obtiene de valor salida al ejecutar el programa y lo que se debería obtener según la especificación** del programa.

Existen varios métodos para comprobar la corrección de un programa, entre los que destacan por ser muy utilizados métodos formales lógicos-matemáticos. Estos métodos requieren un conocimiento previo de sistemas formales de lógica, por lo que no los utilizaremos.

Uno de los métodos más comunes y básicos consiste en **dar distintos y variados valores a las variables de entrada y comprobar que se obtiene un valor de salida esperado de acuerdo con la especificación** o descripción del funcionamiento del programa. Cabe añadir que **los datos de entrada que introduzcamos al programa deben cumplir con su precondición y el resultado esperado, con su postcondición**.

Estos valores que nosotros le daremos a nuestro programa para comprobar el correcto funcionamiento del programa se denominan **certificados o casos de prueba**.

A la hora de crear casos de prueba **debemos procurar que los valores de los datos de entrada nos permitan cubrir todas las posibilidades que los valores de los datos de salida puedan abarcar**. Ciertos valores de entrada se podrían agrupar en distintos grupos en los cuales ese conjunto de datos en específico posee una serie de características comunes. Por ejemplo, para un algoritmo que multiplicara dos números enteros X e Y, tres grandes grupos podrían ser: pares con igual signo, pares con distinto signo y pares con al menos un cero. Estos grupos también caracterizan a los datos de salida esperados. Así respectivamente, se esperarían para cada caso de prueba valores positivos, negativos y ceros.

Por ejemplo, si nuestro programa calcula el producto de dos enteros, podríamos definir la función característica de la siguiente manera:

$$x * y \begin{cases} 0 \; si \; x = 0 \vee y = 0 \\ \sum_{|y|} |x| \;\; si \; (x > 0 \; \wedge \; y > 0) \vee (x < 0 \; \wedge \; y < 0) \\ \sum_{|y|} -|x| \;\; si \; (x > 0 \; \wedge \; y < 0) \vee (x < 0 \; \wedge \; y > 0) \end{cases}$$

De esta forma, podemos construir una tabla de **casos de prueba significativos**. Un caso significativo es aquel que **representa a una agrupación en concreto de datos**. Por ejemplo, el par (6,4) es un caso significativo para el grupo “pares de igual signo, ambos factores positivos”, cuyo resultado esperado es un número positivo. Si en una tabla de casos significativos encontramos por ejemplo (3,4) y (1,5), pares del mismo signo, ambos positivos; entonces dos casos de prueba estarán representando al mismo grupo anteriormente citado, lo que significa que uno de los casos de prueba no es significativo.

Por otra parte, los pares (3,4) y (-6, -2) simbolizan dos casos significativos distintos, puesto las características que caracterizan a estos pares son ligeramente distintas, pues aunque ambos pares son de igual signo, en uno ambos son positivos y en otro ambos son negativos.

Si construyéramos una **tabla de casos de prueba significativos**, resultaría de la siguiente forma:

Entrada	Descripción del caso significativo	Salida
(5, 0)	Segundo factor nulo	0
(0, 3)	Primer factor nulo	0
(0, 0)	Ambos factores nulos	0
(1, 7)	Pares de igual signo, ambos factores positivos	7
(-2, -4)	Pares de igual signo, ambos factores negativos	8
(2, -3)	Pares de distinto signo, primer factor positivo, segundo factor negativo	-6
(-4, 5)	Pares de distinto signo, primer factor negativo, segundo factor positivo	-20

Para construir una tabla de casos significativos debemos por cada uno de nuestros casos **indicar los valores de entrada que se usan en nuestro ejemplo** de caso significativo, **hacer una breve descripción argumentando por qué este caso es significativo** y distinto a las demás agrupaciones posibles de datos, y **definir el valor de salida esperado tras la ejecución del programa**.

Hay que tener también en cuenta que **dependiendo de la fórmula empleada, existe un rango de valores que pueden llevarnos a errores de cálculo** puesto no podemos obtener un número de, por ejemplo, una división entre 0, **o errores relacionados con la asignación u obtención de un valor fuera de contexto**, como por ejemplo, intentar dar como dato de medida una longitud negativa.

Estos supuestos casos que se denominan **excepciones** no los trataremos en este curso, y asumiremos que el quien usa nuestro programa es consciente de que realizar este tipo de operaciones dará un error y corromperá el programa. De todas formas, **la precondición ya nos limita el rango de valores de entrada que podemos utilizar**, y **utilizar un valor fuera del rango de la pre exime de responsabilidad al programador sobre lo que se pudiera obtener de salida** y por tanto, no aporta importancia a la hora de verificarlo. Luego, es obligatorio que los valores de nuestros casos de prueba se encuentren en el rango de la pre.

Por ejemplo, en un programa que calcula dado un valor x la imagen de la función radical $f(x) = \sqrt{1-x}$ podemos intuir que esta función de variable real tiene un dominio determinado de valores, pues no puede tomar valores que hagan que el radicando sea un número negativo, ya que no existe la raíz cuadrada real de un número negativo.

Luego, la precondición del programa debe ser {1-x >= 0} o más bien {x <= 1}

Por lo tanto, a la hora de definir los grupos, destacaremos los valores pertenecientes al rango de la pre y los que no, pero nuestro único caso significativo será el valor de entrada elegido que cumple con su precondición. El resto de casos que no cumplen la condición de entrada, no son válidos. Dentro de los que sí la cumplen, podríamos clasificar los datos de entrada entre los negativos, el cero y los positivos.

Entrada	Descripción del caso significativo	Salida
-3	Números negativos	2
0	El cero	1
1	Números positivos hasta el 1	0
~~2~~	~~Resto de números positivos~~	~~i~~

Una vez definidos lo que son los casos de prueba y cómo se establecen, nos debemos cuestionar la siguiente pregunta. ¿Cómo puedo asegurarme con un alto nivel de certeza que mi programa es correcto? **Consideraremos que un programa es correcto si a la hora de ejecutarlo dando como parámetros de entrada los valores de entrada que le hemos dado a los casos de prueba significativos obtenemos como resultado la salida esperada según la tabla de casos**, y se repite este paso **para todos los casos significativos de la tabla**, asegurándose de que en todos se obtenga la salida esperada. Si en sólo uno de los casos no se obtiene el valor de salida esperado, el programa es incorrecto.

2.3 Ejercicios propuestos

Se sugieren realizar los siguientes ejercicios para comprobar que se han comprendido los conceptos más importantes de este tema. Aunque estos ejercicios no sirvan para programar como tal, sí que forman parte de unos conocimientos previos que hay que tener dominados antes de empezar a programar de verdad.

Para los siguientes enunciados, diseña un diagrama de flujo o algoritmo en pseudocódigo. Si el enunciado es ambiguo, dé al menos dos interpretaciones del mismo:

29. (*) El programa calcula la raíz cuadrada de x menos y
30. (*) El programa calcula el área de un triángulo equilátero dado el lado
31. (*) El programa calcula la diferencia de dos números
32. (**) El programa devuelve una letra correspondiente al día de la semana en castellano dado un número del 1 al 7 correspondiente a la posición del día en la semana.
33. (**) El programa lee por el teclado una secuencia ordenada de menor a mayor de N números (parámetro de entrada por memoria) e imprime el resultado de la media de esos números
34. (**) El programa recibe una fecha mediante dos parámetros, día y mes, e indica la estación del año en la que se encuentra ese día. Consideraremos que la primavera comienza el 21-MAR; el verano, 21-JUN; el otoño, 21-SEP; y el invierno, 21-DEC.

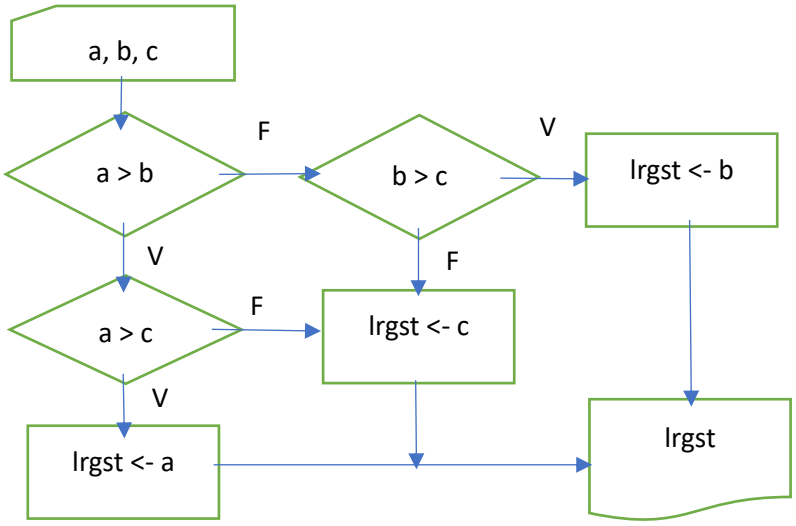
35. (***) El programa escribe en pantalla con cifras los siguientes números: diez, noventa y ocho, ochenta y dos, sesenta y seis, treinta y doce

Para los siguientes enunciados, realiza la especificación pre-post de los siguientes programas. Si el enunciado es ambiguo, dé al menos dos especificaciones del mismo:

36. (*) El programa imprime en pantalla “Hola Mundo!”
37. (*) El programa calcula la suma de los N primeros naturales, empezando por el 1
38. (*) El programa calcula la división de dos enteros
39. (**) El programa recibe mediante entrada estándar una secuencia de caracteres alfanuméricos e imprime en pantalla el nº de apariciones que tiene lugar cada vocal en la secuencia
40. (**) El programa transforma una medida temporal dada únicamente en horas a una medida temporal dada en horas, minutos y segundos
41. (**) El programa decide si un número es primo. Un número N es primo cuando sus únicos divisores son 1 y N
42. (***) El programa decide si un número es pseudoprimo. Un número N es pseudoprimo cuando tiene otros dos divisores distintos de 1 y N y estos divisores son primos
43. (***) El programa recibe la amplitud de tres ángulos y decide a partir de dichos datos si estos lados forman un triángulo o no. En caso de formar un triángulo, decide si este es rectángulo, acutángulo u obtusángulo
44. (***) El programa recibe la longitud de tres lados y decide a partir de dichos datos si estos lados forman un triángulo o no. En caso de formar un triángulo, decide si este es equilátero, isósceles o escaleno
45. (****) El programa calcula de 20 patos metídos en un cajón cuántas patas y picos son

Para los siguientes programas diseñados mediante diagrama de flujo o pseudocódigo, realiza su especificación pre-post:

46. (***)



47. (****)

```
x = 1;   fact := 1;
while x <= n do
   fact := fact * x;
   x := x + 1;
```

48. (****)

```
for i in range 1 to 100 do
   if es_primo(i) then print('*');
```

49. (****)

```
cnt = 0;
repeat
   n = n / 10;
   cnt = cnt + 1;
until n=0;
```

50. (*****)

```
scan(letra);
while letra<>' ' do
  if (letra='x') then print('a');
  else if (letra='y') then print('b');
  else if (letra='z') then print('c');
  else print(next(next(next(letra))));
```

Para los siguientes programas que siguen las siguientes especificaciones, define una serie de casos de prueba significativos:

51. (*) El programa calcula la división de dos enteros X e Y (Ej. 38)
52. (*) El programa calcula el área de un círculo (Ej. 5)
53. (*) El programa resuelve una ecuación cuadrática (Ej. 6)
54. (**) El programa devuelve una letra correspondiente al día de la semana dado un número del 1 al 7 correspondiente a la posición del día en la semana (Ej. 32)
55. (**) El programa recibe una fecha mediante dos parámetros, día y mes, e indica la estación del año en la que se encuentra ese día. (Ej. 34)
56. (**) El programa imprime dada por el usuario una nota de un control, se califique como SUSPENSO (<5), APROBADO (5-7), NOTABLE (7-9) o SOBRESALIENTE (>9) y se muestre en pantalla (Ej. 11)
57. (**) El programa transforma una medida temporal dada únicamente en horas a una medida temporal dada en horas, minutos y segundos (Ej. 40)
58. (**) El programa imprime todos los múltiplos de 3 y de 5 hasta cierto supremo dado por el usuario (Ej. 13)
59. (***) El programa decide si un número es pseudoprimo. (Ej. 42)
60. (***) El programa solicita por el teclado una secuencia de números naturales y se muestre en pantalla, cuando se lea el valor -1, la suma de todos ellos. (Ej. 16)
61. (***) El programa recibe la longitud de tres lados y decide a partir de dichos datos si estos lados forman un triángulo o no. En caso de formar un triángulo, decide si este es equilátero, isósceles o escaleno (Ej. 44)
62. (****) El programa lee por el teclado un número cualquiera, guarda el número de dígitos pares que contiene y la longitud de la secuencia más larga (Ej. 27)

BLOQUE B: INTR. A LA CIENCIA DE DATOS

3. Bases del discurso: Tipos de datos

La palabra **dato** procede de un antiguo participio pasivo del verbo latino *dare*, o en castellano, dar; por lo que como es lógico, *datum* se podría transcribir como lo dado o lo que se da, por lo que podremos definir dato como **la información que se da sobre un suceso concreto y que permite conocer el conocimiento exacto del suceso o al menos deducir sus consecuencias**.

En el ámbito de la Informática, tiene una acepción bastante más acertada, y es: "**información dispuesta de manera adecuada para su tratamiento por una computadora**", y no sólo eso, sino que también esta palabra ha creado nuevas expresiones dentro de ese ámbito y que hoy en día se escuchan casi cada día en la sección Tecnología de nuestro telediario como "base de datos", "gestión de datos", "inteligencia de datos", "minería de datos", "protección de datos", y un largo etcétera, destacando entre ellos Ciencia de Datos.

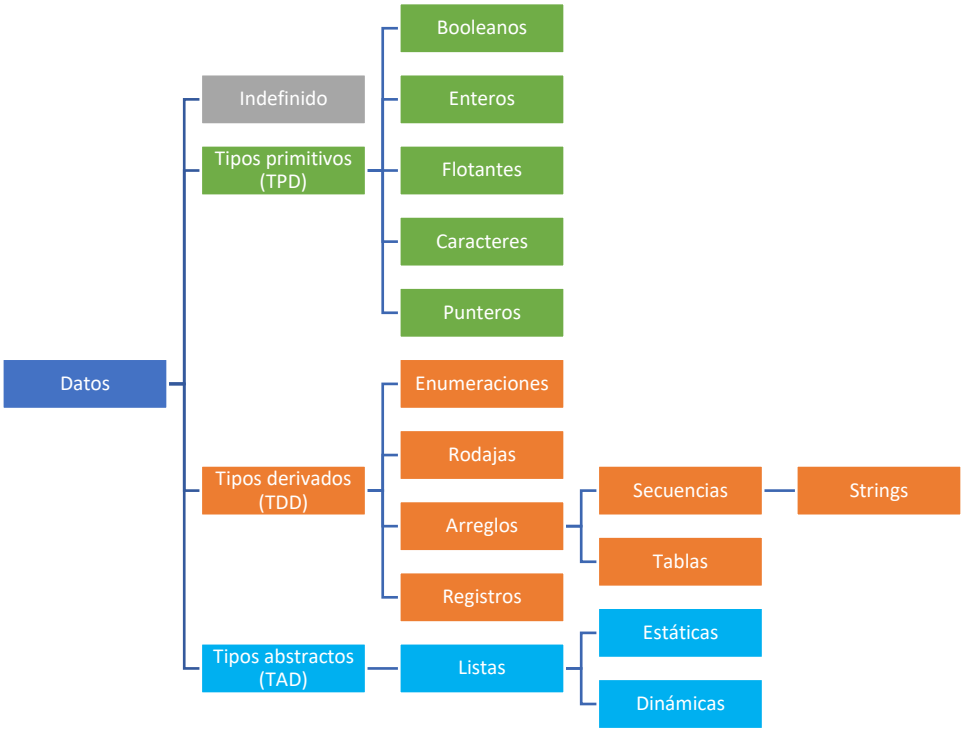
La **Ciencia de Datos** es una de las disciplinas del futuro, más bien es un **campo interdisciplinario** que utiliza diferentes disciplinas como **la estadística, la computación o la algorítmica** para tratar con los distintos datos que se nos pueden presentar en un escenario determinado. Para ello, es muy importante **analizar con detenimiento los datos y estructurarlos**, es decir, clasificarlos.

En la disciplina informática, **el tipo de dato se suele definir como el atributo que indica al programador sobre la clase de datos que se va a manejar**. Esto incluye imponer **restricciones** en los datos, como qué **valores** pueden tomar y qué **operaciones** se pueden realizar.

Conforme a la clasificación de los datos existen tres clases principales:

- ***Tipos primitivos de datos (TPD):*** son datos muy básicos y que forman en sí una **unidad de información atómica**, una unidad mínima de almacenamiento. En esta clasificación pertenecen los **caracteres alfanuméricos**, los **booleanos** (valores de verdad), los **nº enteros**, los **nº reales** y los **punteros**.
- ***Tipos derivados de datos (TDD):*** son datos **creados a partir de la composición de tipos primitivos y estructurados de una forma** determinada. Destacan las **secuencias** o vectores, las **tablas** o matrices, las **enumeraciones**, las **rodajas** y los **registros**. La disposición de los datos primitivos que forman parte del tipo derivado se denomina estructura de dato.
- ***Tipos abstractos de datos (TAD):*** son **datos definidos mediante un modelo matemático y compuestos por un conjunto de datos** más simples **y una serie de operaciones** que se pueden realizar con ellos. **El comportamiento de estos datos no lo define el programador, sino el usuario**. Un buen ejemplo de ello son las **listas**. La implementación de este tipo de datos en un lenguaje de programación se realiza mediante una estructura de datos.

En el contexto de los lenguajes de programación **débilmente tipados**, se podría hablar del **tipo de dato indefinido**. Este tipo de dato surge puesto en este tipo de lenguajes no se declaran previamente las variables, y se declaran inmediatamente tras su primera aparición en el código del programa, pero se tipan una vez se les haya conferido un valor determinado. Si operamos con este tipo de datos, surgirá un error de compilación.



3.1 Tipos Primitivos de Datos

Los **tipos primitivos de datos** se pueden clasificar en:

1. ***Booleanos:*** nombrados así en honor a George Boole, primer matemático que materializó las variables lógicas (que toman sólo el valor de verdadero o falso) a los circuitos eléctricos, **asignando el valor 1 (TRUE) como encendido, y el valor 0 (FALSE) como apagado**. No existen bombillas encendidas y apagadas a la vez ni enunciados atómicos verdaderos y falsos a la vez. Es un **valor discreto y binario que indica la veracidad de una propiedad**. Es un sí o un no, un ying o un yang.
2. ***Enteros:*** se denominan así porque **representan cantidades exactas**. Son los números **utilizados para contar**, por ejemplo para indicar los pisos que tiene un edificio, incluso los negativos, indicándote que se trata de uno subterráneo. Sería un sinsentido tener un rebaño de 6.22 ovejas o un andén 9 y ¾ salvo que vivas en el universo literario de J.K. Rowling. Aunque el concepto de número no existió hasta las primeras civilizaciones, se ha demostrado que varios animales saben contar, lo que indica que **el conteo es propiedad intrínseca de la naturaleza**. Y por si esto fuera poco, en el año 5.000 a.C. en Sumeria se creó la primera computadora del mundo, un ábaco capaz de realizar todo tipo de operaciones, incluso con números negativos, basado eso sí en el conteo de cálculos o piedras pequeñas. Se le considera de hecho el predecesor de la calculadora.

3. ***Flotantes:*** son aquellos números que **no representan cantidades exactas, sino fracciones, partes o porciones** de una unidad como una tarta, una pizza o un **porcentaje** de una muestra poblacional, o una **aproximación de un valor irracional**, el cual no puede representar como fracción. **Los números enteros también están incluidos entre los números reales**, ya que ocho porciones de una pizza formarían la unidad entera, o de manera más formal, cualquier entero se puede representar como el cociente de dos reales, P.ej. 2 = 2/1 = 4/2 = -4/-2 = 1/0.5 = -1/-0.5, etc. Aunque cabe aclarar que **lo que se denomina número real en Matemáticas no corresponde exactamente con lo que se define en Informática como su tipo de dato correspondiente**, puesto la memoria tiene una capacidad finita y no se pueden guardar ni números muy muy grandes ni números muy muy pequeños ni todos los decimales del número π, que de él hasta ahora se han calculado 62 billones, necesitamos una aproximación decimal (lo que implica que se puede representar en formato racional), es lo que se le conoce en Informática como **coma flotante**.
4. ***Caracteres:*** todos aquellos **caracteres alfanuméricos** que no sirven ni para contar, representar porciones, aproximar valores reales o indicar la veracidad de una propiedad son los caracteres propiamente dichos, entre los que incluyen **las letras, los signos, los símbolos y los propios dígitos** del 0 al 9 usados para escribir los números. Conjuntos de caracteres forman la composición íntegra de esta obra.
5. ***Punteros:*** dependiendo del manual del lenguaje de programación con el que se trabaje, a veces se le considera como un tipo de dato primitivo y otras veces no. Realmente es un **"objeto" de referencia cuyo valor se refiere a otro valor almacenado en otra parte de la memoria del ordenador utilizando su dirección**. El puntero **apunta a una ubicación** en memoria, y a **la obtención del valor almacenado en esa ubicación se la conoce como desreferenciación** del puntero. Análogamente, el número de página en el índice de este libro podría considerarse un puntero a la página correspondiente; al ir a la página con el número de página especificada en el índice habremos desreferenciado el puntero. No entraremos en mayores detalles sobre este tipo de dato en concreto, se verá más adelante en el capítulo 10.

Por último, cabe aclarar que los distintos lenguajes de programación pueden considerar más o menos tipos primitivos de datos de los que se exponen según ellos consideren como lo básico.

Por ejemplo, en C no existe como tipo primitivo los booleanos por lo que en su lugar debe usarse un Integer que tome los valores 0 y 1 en sustitución de false y true. De hecho, los lenguajes débilmente tipados no tienen en su gramática el tipo de dato primitivo "puntero". Sin embargo, C tiene otros tipos de dato primitivo como los "unsigned character" (variable que reserva 16 bits de memoria codificados en sistema hexadecimal)

El tipo de dato String o cadena de caracteres se considera primitivo hoy día por la mayoría de lenguajes de programación. Sin embargo, en este libro se considerará como los antiguos lenguajes, como tipo derivado.

Otros lenguajes como Java tienen algún tipo de dato duplicado como los Float, pero cuya diferencia es el espacio de memoria que ocupan, de esta forma “short” indica los reales de capacidad 16 bits, “float” los reales de capacidad 32 bits, y “double” los reales de capacidad 64 bits. Recordemos que la unidad mínima de información del ordenador es el bit, cuyo valor sólo comprende entre 0 y 1.

A continuación, se exponen las distintas **operaciones que podemos realizar con los distintos TPDs** y cómo los indicaremos en pseudocódigo y en los distintos lenguajes de programación que se desarrollarán a lo largo de este libro.

Con los booleanos, podemos realizar las operaciones lógicas de **negación** (not) (cambiar de true a false y viceversa), **disyunción** (or) (devuelve true si uno de los dos operandos es true) y **conjunción** (and) (devuelve true si y solo si ambos operandos son true) Aquí se muestran las tablas lógicas de las tres operaciones, que indica el valor de la operación según los valores de las variables booleanas a, b:

a	not a
true	false
false	true

a	b	a or b
true	true	true
true	false	true
false	true	true
false	false	false

a	b	a and b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a xor b
true	true	false
true	false	true
false	true	true
false	false	false

Cabe destacar de que ADA tiene una operación extra denominada **disyunción exclusiva** (xor), la cual devuelve true si uno de los dos operandos es true, pero no ambos. Se podría utilizar también en pseudocódigo. Se muestra a la derecha su tabla de verdad.

a xor b equivale a la expresión (a or b) and not (a and b)

Por otra parte, los operadores en C son radicalmente distintos a los anteriormente citados (not, or, and), que son compatibles tanto para pseudocódigo como para Pascal, ADA y Python.

- not a → ! a
- a or b → a || b
- a and b → a && b

El tipo de dato booleano sólo puede tomar dos valores: true y false, que a veces, como en C, ya que en C no existe el tipo booleano, puede escribirse como 1 y 0, respectivamente.

Con los enteros, destacan dos tipos de operaciones. El primer tipo se corresponde con las **operaciones aritméticas**, y estas son **la suma (+), la resta (-), el producto (*), la división entera (/)** (que devuelve el cociente de la división), **el módulo (%)** (que devuelve el resto de la división*), **la exponenciación (^) y el valor absoluto (abs)**. Utilizaremos esta notación para expresar las siguientes operaciones en el pseudocódigo.

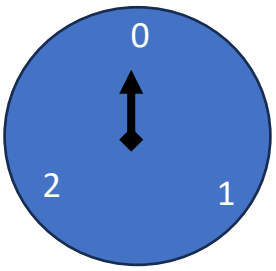
En Pascal y en C, la exponenciación y el valor absoluto no están incluidos en la gramática del lenguaje. A diferencia de C, Pascal utiliza como operadores para la división entera y el módulo (div, mod), respectivamente.

Python sí tiene exponenciación, y para ello utiliza el operador **, al igual que en ADA, pero no operador de valor absoluto, cosa que ADA sí tiene. Por otra parte, la división entera la indica con una doble barra (//), que es la que devuelve sólo el cociente de la división. Si utilizáramos la barra simple en Python (/), se convertiría en una división común, lo que hará Python será obtener el resultado exacto de la división, decimales incluidos.

Por otra parte, **ADA realiza una distinción entre el resto de una división y el módulo de un número en una base**, relacionado esto último con la **aritmética modular**, bajo los operadores rem y mod, respectivamente. **Estas operaciones son idénticas si los operandos tienen el mismo signo***, pero si estos tuvieran distinto signo, se distingue mejor la diferencia entre resto y módulo.

Por ejemplo, para x rem 3, los restos de estas divisiones son los siguientes: Recordemos que una división entera a / b debe cumplir la propiedad a = b * q + r, siendo a el dividendo, b el divisor, q el cociente y r el resto, r < b.

x	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
x rem 3	-1	0	-2	-1	0	-2	-1	0	1	2	0	1	2	0	1



Y para x mod 3, se podría asimilar dicha operación como si fuera un reloj de tres horas; 0, 1 y 2 y mover la aguja empezando desde el cero x veces hacia la derecha si el número es positivo y x veces hacia la izquierda si el número es negativo. La hora que marque la aguja será el resultado de la operación.

x	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
x mod 3	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1

Luego, **las operaciones "a rem b" y "a mod b" dan el mismo resultado cuando "a, b" tienen el mismo signo o "a" sea múltiplo de "b"**. Por ello, cuando nos pidan por ejemplo un número par, la condición que debe cumplir puede ser perfectamente o "a rem 2 = 0" o "a mod 2 = 0", o en otros lenguajes, "a%2 = 0"

Cabe destacar que **no todas las operaciones aritméticas que se nos ocurran** (p.ej. raíz cuadrada, raíz quinta, logaritmo, seno e incluso exponenciación en algunos lenguajes etc.) **están disponibles en los lenguajes de programación**, por lo que para utilizar una operación no existente en dichos lenguajes **debemos o bien utilizar un algoritmo** que defina dicha operación, **o bien instalar el paquete de recursos en el cual esa operación está incluida como una función** (p.ej. sqrt(), exp(), log(), sin(), etc.), suelen tener el nombre de Maths o Numerics. Se especificará siempre en el enunciado si hay que importar alguna librería y cómo se hace según el lenguaje de programación que se use.

El segundo tipo de operaciones corresponde con las **operaciones relacionales**. En pseudocódigo y en Pascal, estas son: **mayor que** (>), **mayor o igual que** (>=), **menor que** (<), **menor o igual que** (<=), **igual a** (=) y **distinto de** (<>)

En el resto de lenguajes, los cuatro primeros operadores son idénticos, para la operación "distinto de", su operador en ADA es (/=), mientras que en Python y C es (!=); y para la operación "igual a", su operador en Python y en C es (==)

Esto es debido a que la asignación en estos dos lenguajes utiliza como operador (=), a pesar de encontrarse con la oposición de varios programadores, ya que cuando se asigna un valor a una variable, no se iguala la variable a dicho valor, sino que guarda en una dirección de memoria con el nombre de esa variable ese valor.

Luego, hay que tener cuidado de no equivocarse con la operación "igual a", la cual devuelve siempre un valor booleano, y la asignación. Si la comparación de igualdad se expresa mediante el operador (=), entonces el operador de asignación será (:=), es lo que sucede con Pascal y ADA. Si pasa al contrario como en Python y C, en el que el operador (=) es el de la asignación, entonces, la operación "igual a" tendrá como símbolo (==)

Matemáticamente, el conjunto de los números enteros es infinito, abarca desde el – ∞ hasta el + ∞, pero en la Informática, **el tipo de dato entero está acotado entre dos números**: uno muy muy pequeño y otro muy muy grande**. Esto es debido a que la memoria tiene una capacidad finita** y no se pueden almacenar ni números muy muy muy pequeños ni números muy muy muy grandes.

Estos extremos están por defecto etiquetados en los lenguajes de programación. Por ejemplo, para Pascal, [MinInt, MaxInt]; para Ada, [Integer'First, Integer'Last]; para Python, [sys.minint, sys.maxint] (debe haberse instalado previamente la librería sys): y para C, [INT_MIN, INT_MAX] (debe haberse instalado previamente la librería limits.h)

Dependiendo de la **arquitectura del procesador** (es decir, el número máximo de bits que puedo almacenar en memoria para el valor de una variable), el valor de estos extremos es para MinInt = -2^31 o -2^63 y para MaxInt = 2^31 – 1 o 2^63 – 1, según la arquitectura de 32 bits y la de 64 bits, respectivamente.

Como dato adicional, no viene mal saber que **se pueden utilizar barras bajas (_) como separador de dígitos**, realizando la misma función que el punto que separa millares en español, o la coma en inglés. En lugar de escribir 1,234,567 podemos escribir 1234567 o 1_234_567.

Con los flotantes, o vulgarmente dicho, los reales, **podemos usar todas las operaciones que se habían definido anteriormente para los enteros, a excepción de la división entera y el módulo, que se sustituyen por la división normal (/)**. Esta división da el resultado exacto con precisión decimal de dividir a entre b, devolverá un número real k que multiplicado por b diera como resultado a.

Conforme a su representación, surge el mismo problema de memoria, pero esta vez **deben guardarse tanto los dígitos de parte entera como los dígitos de la parte decimal** de nuestro número real, y distinguirse entre sí. **Esta manera de representar reales por los valores de sus parte entera y de su parte decimal se denomina coma flotante**, puesto en realidad, el número se representa a través de una mantisa (el número decimal sin la coma), multiplicado por una potencia de base 10

elevada a un exponente determinado. Este exponente decide cuántas posiciones a la izquierda o al a derecha se desplaza la coma con respecto al último dígito donde termina la parte entera del número, de forma que quede sólo un dígito del 1 al 9 delante de la coma (cuidado, en inglés, nuestra coma decimal se representa mediante un punto), de **una forma parecida a la notación científica** en la calculadora.

P.ej. 3234.5 → 3.2345E+3, 0.00016 → 1.6E-4, que sustituyen a las formas de representación matemáticas 3.2345 * 10^3 y 1.6 * 10^-4, respectivamente.

Evidentemente, podemos escribir cualquier número decimal usándolo como es en su esencia, con su parte entera, su coma decimal y su parte decimal. Sin embargo, hay que tener en cuenta de que **si por alguna razón necesitáramos asignarle un valor entero a una variable flotante, es obligatorio escribir el número con la coma decimal** para indicar que su tipo de dato es el flotante y no confundirlo con un entero. Luego, en lugar de asignar a una variable el valor 5, se le asignará el valor 5.0.

Como máximo, a la hora de asignar valor a las variables flotantes, se pueden utilizar números de hasta 24 dígitos (de los cuales 7 son de la parte decimal) para sistemas de 32 bits; y números de hasta 53 dígitos (de los cuales 15 son parte decimal) para sistemas de 64 bits.

Finalmente, tenemos **los caracteres**, que abarcan desde espacios en blanco, puntos, comas, signos, símbolos, hasta las letras que comúnmente conocemos del abecedario latino, de la A a la Z tanto en minúsculas como en mayúsculas.

Las operaciones que se pueden realizar con caracteres **son más limitadas, y no son nada similares a las operaciones matemáticas que conocemos**, por lo que es cuestión de práctica el tener que acostumbrarse a trabajar con caracteres.

Para ello, se liga mediante un **código internacional** llamado **ASCII** cada carácter con un valor numérico determinado. Un anexo aparte mostrará algunos de los **256 caracteres ASCII los cuales están ligados a un índice cada uno**. Así por ejemplo, la letra 'A' está ligada al índice 65.Es importante fijarse en esta relación entre carácter e índice (no memorizarla, se dará el correspondiente ASCII del carácter en el enunciado), ya que entonces, se entenderá mucho mejor la aritmética de caracteres. **A la hora de denotar caracteres, los denotaremos entre comillas simple**s, para no confundir la variable a con el carácter 'a'

En primer lugar, viene bien saber que **se pueden utilizar operaciones relacionales en todos los lenguajes de programación. El orden de caracteres lo establece el propio código ASCII**, y tiene la ventaja de ser intuitivo, puesto **se sigue el orden numérico para los dígitos y el alfabético para las letras**, tanto mayúsculas como minúsculas, aunque siendo "mayores" las letras minúsculas al aparecer después de las mayúsculas.

Algunos enunciados booleanos verdaderos con estos operadores relacionales son p.ej. 'a' = 'a'; 'a' <> 'A'; 'a' > 'A'; 'A' < 'a'; 'b' < 'g'; 'z' > 'm'; '3' < '4'; '5' > '0' y otros enunciados menos intuitivos como '!' < '+'; '&' > '$'; "" <> ""; '2' > '%'; '7' < '^'; '0' < 'o'; 'D' < 'c': 'd' > 'c'; ' ' <> ''.

Otras operaciones que pueden aplicarse a los caracteres son: **el ordinario de un carácter** (ord()), que devuelve el índice ASCII ligado al carácter; su inverso, **el carácter de un número** (chr()), en el que

dado un número dentro del rango de índices ASCII, devuelve el carácter correspondiente según el índice; las operaciones booleanas **isAlpha(), isDigit(),** en las que dado un carácter, indica si este es una letra del alfabeto o un dígito numérico, respectivamente; y las operaciones **upper(), lower(),** que transforman una letra del alfabeto a mayúscula y a minúscula, respectivamente. En algunos lenguajes y en pseudocódigo, existen dos operaciones extra válidas**, succ() y pred(),** en las que dado un carácter, devuelve el siguiente y el anterior carácter al dado, respectivamente.

Por ejemplo, ord('B') = 66, chr(66) = 'B'; isAlpha('B') = true; isDigit(`B`) = false; upper('B') = 'B'; lower('B') = 'b'; succ('B') = 'C'; pred('B') = 'A'.

Evidentemente, existen bastantes más caracteres que los dados, pero nos vamos a abstener a utilizar exclusivamente los caracteres del inglés, ya que **existen muchos lenguajes que no aceptan caracteres especiales**, es decir, caracteres de otras lenguas distinta a la inglesa. Luego, a la hora de crear nuestros programas, **debemos procurar no utilizar caracteres especiales, lo que incluye caracteres comunes del español como letras con tilde, la u con diéresis y la letra eñe**, por lo que les suprimiremos sus tildes, diéresis y virgulillas. Así, en vez de escribir á, é, í, ó, ú, ü, ñ; escribiremos a, e, i, o, u, u, nn**. Los signos de apertura de interrogación (¿) y de exclamación (¡) simplemente se omiten**.

Conforme a los Strings, en este libro se les considerarán como una secuencia de caracteres y no como un TPD, por lo que se impartirán en el Tema 8: Arrays.

Los punteros también son otro TPD, pero se impartirán mucho más adelante, cuando se usen en nuestros programas más sofisticados y con estructuras de datos más complejas.

Las distintas operaciones para los distintos TPDs **pueden combinarse entre sí siempre y cuando los dos operandos resultantes no sean de distinto tipo**. P.ej, 'A' + 3 es una operación inválida pero ord('A') + 3 sí lo sería, pues el ordinario de 'A' devuelve un resultado del mismo tipo de dato que 3, un entero. De la misma forma, 3 + 5.2 sería inválido, pero 3.0 + 5.2 sí lo sería. **También sería inválido operar con un tipo de dato que ese operador concreto no admite**, por ejemplo, expresiones como succ(true), 'd' – 'c', 3 or 4 son inválidas.

La **jerarquía de operaciones** que se sigue al operar expresiones combinadas es la siguiente:

1. Paréntesis
2. Exponenciación y funciones matemáticas
3. Operadores de caracteres (y enumeraciones, rodajas)
4. Desreferenciación (punteros y accesos de arreglos o registros)
5. Negación
6. Producto, división, módulo y conjunción
7. Suma, resta y disyunción
8. Operadores relacionales
9. De izquierda a derecha

P.ej:	5 * (3 + 4) = ord('B') / 2 + 2 or not (5^2 > sqrt**(ord('d'))** and isDigit**(chr(50))**)
Paréntesis	5 * 7 = ord('B') / 2 + 2 or not (**5^2 > sqrt(100)** and isDigit('2'))
Exponenciación y funciones mat.	5 * 7 = **ord('B')** / 2 + 2 or not (25 > 10 and **isDigit('2')**)
Operadores de caracteres	5 * 7 = 66 / 2 + 2 or not (25 > 10 and true) = 5 * 7 = 66 / 2 + 2 or **not true** {Por preferencia del paréntesis}
Negación	5 * 7 = 66 / 2 + 2 or false
Producto, división, módulo y conjunción	35 = 33 + 2 or false {La disyunción depende del valor de la expresión 35 = 33 + 2}
Suma, resta y disyunción	35 = 35
Operaciones relacionales	true {Resultado final}

Para finalizar, se muestran a modo de resumen los distintos operadores que hemos visto para manejar diferentes TPDs para pseudocódigo y para los lenguajes de programación con los que trabajaremos en otro anexo aparte.

3.2 Tipos Derivados de Datos

Los **tipos derivados de datos** se pueden clasificar en:

1. ***Enumeraciones:*** es un tipo de dato compuesto por varios elementos llamados **enumeradores o identificadores**, los cuales están **etiquetados**. Por ejemplo, una enumeración de los posibles mazos que pueden obtenerse en una carta de la baraja española es Mazos = (Oros, Copas, Bastos, Espadas), y la de sus posibles valores es Valores = (As, 2, 3, 4, 5, 6, 7, Sota, Caballo, Rey) Como se puede observar, **la enumeración actúa como una variable qué solo puede tomar como valores un conjunto de elementos** ordenado o no, finito y determinado por el usuario, por ello, a los TDDs se les conoce también como tipos definidos por el usuario. **Esto confiere mayor seguridad a las variables**, ya que si por ejemplo, declaro Sexo como el TPD carácter, se espera que el usuario introduzca 'M' para masculino y 'F' para femenino, pero puede introducir otro carácter distinto que no nos interese. Por ello, declaramos Sexo como el TDD enumeración, para limitar el nº discreto de valores que puede tener una variable de ese tipo a M, F (ojo, aquí son etiquetas, no caracteres). Si entonces se introdujera otra etiqueta, entonces se imprimirá en pantalla un mensaje de error avisándole al usuario de que se ha equivocado. Normalmente, las enumeraciones agrupan una serie de identificadores que comparten algo en común, p.ej. Sexo = (Masculino, Femenino), Estaciones = (Otoño, Invierno, Primavera, Verano) Además, podemos utilizar como etiquetas un número, una letra, una palabra o una serie de palabras, a nuestra libre disposición, llegando a mezclar dichos formatos de etiquetado incluso para la misma enumeración.
2. ***Rodajas:*** aunque sólo están disponibles como TDD ya implementado para algunos lenguajes, en este libro introduciremos este concepto a nuestro pseudocódigo ya que es una óptima herramienta para los principiantes en la programación que ayuda a simplificar el código. En otros manuales, esta herramienta no se reconoce y se opta por diseños más formales y

concurrentes típicos de la mayoría de lenguajes. La rodaja es un tipo de dato que **deriva de un rango de valores perteneciente a un tipo primitivo**. El nombre rodaja sugiere como si de tener un solomillo y tomar la parte del solomillo que más nos gustara se tratara. **En este símil, el solomillo entero es el TPP**, mientras que **la porción de solomillo** que queremos servir en nuestro plato **es la rodaja**. Por ello es que también se le conoce como **subrango o subtipo**. Por ejemplo, el conjunto de los nº naturales es una rodaja o subconjunto del conjunto de los nº enteros. Sabemos que todos los naturales son enteros, pero no todos los enteros son naturales. Por lo tanto, decimos que los naturales son un TDD rodaja del TPD entero. Así mismo, el número 3 pertenece tanto al TPD entero como al TDD natural, y por lo tanto, **las operaciones que se puedan realizan en el TPD también pueden realizarse sobre el TDD siempre que su resultado quede dentro del rango**. Las rodajas se indican bajo la notación a..b, siendo a el primer valor del subrango y b, el último. De esta forma, siendo el rango de los enteros int = MinInt..MaxInt, el subrango de los naturales se denotará como nat = 0..MaxInt.

3. ***Arreglos:*** es un tipo derivado que almacena **una serie de valores contiguos del mismo tipo**. Se subdividen en dos tipos principales: las **secuencias o vectores**, cuando este arreglo está organizado en una única fila como por ejemplo el vector u; y las **tablas o matrices**, cuando este arreglo está organizado en filas y en columnas como por ejemplo la matriz A. Sus nombres de variable se denotan con uno o dos parámetros de dimensión encerrados entre corchetes a su derecha: u[n] y A[m, n], siendo m el número de filas que posee el arreglo (en el caso del vector, m=1, se omite) y n el número de columnas. A la hora de asignar los valores del arreglo, estos deben ir entre corchetes y separados por comas. En el caso de las matrices, cada fila debe ir separada por punto y coma. En este ejemplo, u[3] = [3, 1, -1] y A[3,2] = [1, 2; 0, -1; 3, 1] Debajo se muestran lo que serían sus equivalentes en notación matemática:

$$u = (3 \quad 1 \quad -1); A = \begin{pmatrix} 1 & 2 \\ 0 & -1 \\ 3 & 1 \end{pmatrix}$$

4. ***Registros:*** es un tipo de dato derivado que almacena **una serie de valores contiguos pero que estos pueden ser de diferente tipo**. Este conjunto de valores denominados **campos forman una estructura de datos** que definen un objeto cotidiano. Son **uno de los tipos más comunes que nos podemos encontrar como representación de conceptos más complejos**. P.ej. un documento de identidad, la etiqueta de una prenda, el ticket de la compra, una carta de la baraja española, etc. Para declarar un nuevo tipo de registro, se denota escribiendo cada campo entre llaves, separados por comas. P.ej. ID = {Nombre, Apellidos, Num, Letra}, y para crear un objeto de ese tipo, lo mismo pero dando ya los valores de esos campos. P.ej. ID1 = {“Carmen”, “Muestra Muestra”, 123456789, ‘A’} Como se puede observar, las comillas son imprescindibles, puesto diferencian claramente entre lo que es el nombre del campo, y el valor y tipo de ese campo.

A continuación, se exponen las **operaciones de conjunto** que se pueden realizar con los TDDs enumeración y rodajas, así como con cualquier TPD. Trataremos más adelante y más detalladamente con los TDDs arreglos y registro en sus temas, el 8 y el 10, respectivamente.

Dados los TDDs enumeración Días = (L, M, X, J, V, S, D) y rodaja Alfabeto = ‘A’..’Z’ (compuesta por los caracteres ASCII que pertenecen al alfabeto y son mayúsculos), las operaciones que se pueden realizar son las siguientes:

- ***Operadores relacionales:*** de la misma forma que 'A' < 'F' (al aparecer A antes que F en el alfabeto y en la lista de caracteres ASCII), y 'W' > 'E', con las enumeraciones podemos hacer lo mismo, por lo que L < X, V > M, S <> D, son enunciados correctos. Aunque cabe añadir que no en todas las enumeraciones tiene sentido hablar de orden, p.ej. en la enum. Mazos anterior tiene lógica para el computador enunciados como Oros < Espadas o Espadas > Bastos, porque el orden con el que hemos declarado esta enumeración es Mazos = (Oros, Copas, Bastos, Espadas), pero el orden de mazos en realidad no tiene ningún fin práctico. En el caso de Días, tiene su fin porque sigue el orden de los días de la semana, empezando por el Lunes.
- ***Extremos:*** se realiza bajo las operaciones **first()** y **last()**, que devuelven respectivamente el primer y último valor del conjunto de valores que forma el tipo de dato. Por ejemplo, first(int) = MinInt, first(nat)= 0, first(char)='', first(Días) = L, first(Alfabeto) = 'A', last(Días) = D, last(Alfabeto) = 'Z', last(Mazos) = Espadas.
- ***Posicionamiento:*** la operación **pos()** devuelve la posición relativa de un valor de la enumeración o de la rodaja en el conjunto que lo define. Por ejemplo, pos(L) = 1, pos(S) = 6, pos(V) = 5, pos(Bastos) = 3, pos('L') = 12. Por otra parte, su operación inversa es **val()**. Así para el tipo de dato Día, val(1) = L, val(6) = S, val(3) = X; para el tipo de dato Alfabeto, val(12) = 'L', val(5) = 'E'.
- ***Anterior y posterior:*** se utilizan las operaciones **pred()** y **succ()**, que devuelven respectivamente el anterior y el posterior de un valor. P.ej. pred(0) = -1, succ(0) = 1, pred('C') = 'B', succ('P') = 'Q', pred(S) = V, succ(X) = J, pred(Bastos) = Copas, succ(Oros) = Copas. Hay que tener **especial cuidado con no introducir como operando un extremo del tipo**. Para cualquier tipo T, las expresiones pred(first(T)) y succ(last(T)) darán lugar a un error de desbordamiento. De este modo, expresiones como pred(L) o succ(D) son ilegales.
- ***Pertenencia:*** El operando **(in)** recibe como operandos un valor y el nombre de un tipo de dato y **devuelve un booleano que indica si ese dato pertenece a dicho tipo de dato**. P.ej. estos enunciados que utilizan operador de pertenencia son verdaderos: 3 in nat, 3 in int, -1 not in nat, -1 in int, '&' in char, 'G' in char, '&' not in Alfabeto, 'G' in Alfabeto, Bastos in Mazos, Corazones not in Mazos, J in Días, W not in Días.

Las principales diferencias entre estos cuatro TDDs radican en si se permiten que los tipos de dato que componen estos sean distintos o no y en si almacenan sólo un valor de su rango (valores discretos) o más de un valor (valores contiguos).

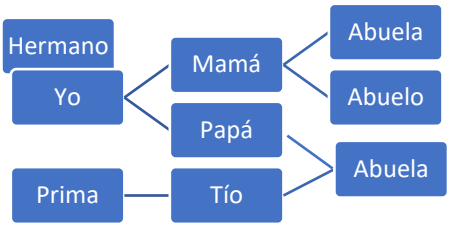
	Valores discretos	Valores contiguos
Valores del mismo tipo	Rodajas	Arreglos
Valores de distinto tipo	Enumeraciones	Registros

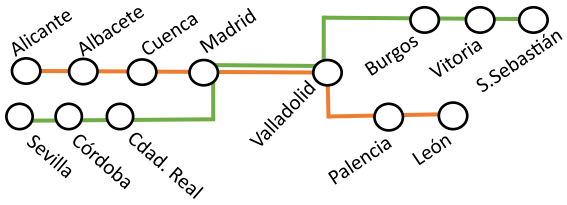
3.3 Tipos Abstractos de Datos

El resto de **estructuras de datos definidas por modelos matemáticos**, siendo estos **definidos a la vez por el punto de vista del usuario** y cómo operan con dichas estructuras, son los denominados **TADs**.

No vamos a desarrollar este tipo de datos en este curso ya que entonces complicaríamos bastante el temario, y este curso se supone que debe introducirnos al mundo de la programación.

Simplemente se deben dar a conocer este tipo de datos para hacer la siguiente objeción: "**Toda forma de organización de los datos que al ser humano se le ocurra es inherentemente un tipo abstracto de dato**" Como ejemplos, se muestran aquí los más comunes, una lista de la compra, un árbol genealógico, un plano de líneas de transporte, la fila del supermercado, etc.





Nos centraremos únicamente y de forma superficial en las **listas**, al ser la estructura más parecida a la secuencia, en el tema 11.

3.4. Ejercicios propuestos

Se sugieren realizar los siguientes ejercicios para comprobar que se han comprendido los conceptos más importantes de este tema. Aunque estos ejercicios no sirvan para programar como tal, sí que forman parte de unos conocimientos previos que hay que tener dominados antes de empezar a programar de verdad.

63. Dadas las siguientes variables a = true, b = false, c = 6, d = -2, e = 0, f = 5, g = 2, h = 0.0, i = 1.0, j = 'y', k = 'e', l = '2', m = '@'; evalúe las siguientes expresiones. Si la expresión produce un error, indícalo y por qué:
    - (*) not a
    - (*) a or b
    - (*) a and b
    - (*) a and not b
    - (*) a + b
    - (*) f + g
    - (**) upper(j)
    - (**) pred(k)
    - (**) isDigit(k)
    - (**) succ(m)
    - (**) c * d + f / g
    - (**) 2.5 * (2 .0 + i)

- (*) c – d
- (*) d * f
- (*) f / g
- (*) f / e
- (*) c % f
- (*) i % -d
- (*) g + i
- (*) 3.0 * i
- (*) h + 2
- (**) ord(j)
- (**) ord('j')
- (**) isAlpha(e)
- (**) isAlpha('e')
- (**) isAlpha(l)
- (**) upper(l)
- (**) a > b
- (**) e < f
- (**) g + c >= 5
- (**) (i + h)/2.0 > 0.5
- (**) 'y' <> j
- (**) m > k
- (**) k < j
- (**) 'k' < 'j'
- (***) (2*c % f = g) xor isDigit(l)
- (***) succ (chr(ord('j') + g))
- (***) upper(pred(k))
- (***) isAlpha(j) and (k < succ('k'))
- (***) 3.5*i + 2.0 > c - d
- (***) not isDigit(chr(ord('+') + f))

- (****) c + d / g – (ord(k) – ord (m)) / c + ord(succ(chr(sqrt(10*g + f) + f*c)))
- (****) (-f^g > -ord(m)) and isAlpha(chr(10^-d + ord(l) + c*d – ord(pred('2'))))

64. Dadas las siguientes variables del tipo enumeración Días = (L, M, X, J, V, S, D): Hoy = X, Ayer = L y de los tipos rodaja Positivo = 1..MaxInt : n = 1, y Alfabeto = 'A'..'Z' : Letra = 'E'; evalúe las siguientes expresiones. Si la expresión produce un error, indícalo y por qué:
    - (*) pred(Hoy)
    - (*) pred(Ayer)
    - (*) succ(Hoy)
    - (*) succ(Ayer)
    - (*) isAlpha(Letra)
    - (*) lower(Letra)
    - (*) n – 1
    - (**) Letra > 'D'
    - (**) Letra < J
    - (**) L < J
    - (**) 'L' < 'J'
    - (**) n in int
    - (**) 'T' in Alfabeto
    - (**) 't' in Alfabeto
    - (**) 'X' in Días
    - (**) X in Días
    - (***) pos('S')
    - (***) pos(S)
    - (***) n + ord(Letra)
    - (***) val(ord(Letra) / 12) para Días
    - (***) val(ord(Letra) / 12) para Alfabeto
    - (***) first(Alfabeto) = Letra
    - (***) last(Alfabeto) <> 'z'
    - (****) pred(val(pos(Hoy) - 1)) para Alfabeto

4. Instruir el discurso: Asignaciones

Empezamos a continuación con lo que se podría denominar la parte nuclear de la asignatura, que comprende básicamente los capítulos 4-8, en donde **diseñaremos ya nuestros primeros programas** totalmente funcionales en los distintos lenguajes de programación. Para ello, usaremos a lo largo de este curso varias herramientas de software para los varios lenguajes de programación ya mencionados.

No obstante, cabe destacar que este libro no utiliza la metodología clásica en la que se enseña el lenguaje de programación como un idioma del que se debe obtener un título de nivel desde A1 a C2. Personalmente, esto lo considero un error muy grave, ya que merma la capacidad de razonar del futuro programador, pues se tiende a mecanizar los problemas, ya que en la metodología clásica, se van enseñando poco a poco las sentencias que se pueden dictar con el lenguaje de programación, para qué sirven y cómo aplicarlas, desde conceptos básicos casi comunes hasta conceptos muy nativos pero qué sólo funcionan en ese lenguaje.

El objetivo de esta obra es poder acercarnos a hablar con del ordenador, no tanto su idioma, sino acercándonos a **cómo piensa el ordenador**; por ello el subtítulo no dice "guía para hablar", sino "guía para discursar". Los lenguajes de programación (idiomas) son en realidad implementaciones de los algoritmos (ideas) que hemos estado diseñando a lo largo de esta obra, y el **pseudocódigo** y el **diagrama de flujo** son como dos lenguajes universales comprensibles a nivel humano alrededor de todo el globo terráqueo y que **simulan el pensamiento computacional**, pero **el ordenador no es capaz de comprenderlo**.

La **Programación** es uno de los **tópicos más importantes de la Ingeniería Informática**, así como la **base mínima** que en este siglo de la **digitalización** todo ingeniero debe tener, por no decir que es la llave que abre las puertas entre el mundo exterior y el ordenador. Todo uso que se te ocurra de las Nuevas Tecnologías, en especial de las TICs, o las **Tecnologías de la Información y de la Comunicación**, necesita programas.

Para poder resolver estos grandes problemas, no se consta ni de mecanizar ni de copiar y pegar métodos de los grandes programadores, **la Programación consiste en resolver a través del razonamiento de un ordenador y de forma razonada un problema**, siguiendo las normas gramaticales de un lenguaje determinado y explicar por qué se ha resuelto en esa serie de pasos determinado. De nada sirve talar un bosque si no se sabe hacer leña. Y es que al fin de al cabo, todos los caminos llevan a Roma, puede existir más de una resolución al problema. Cualquier problema que se nos ocurra, lo puede resolver nuestro ordenador siempre y cuando sepamos qué hay que hacer y se lo indiquemos en una forma que lo comprenda, como a un gólem cuando le introducimos un papel.

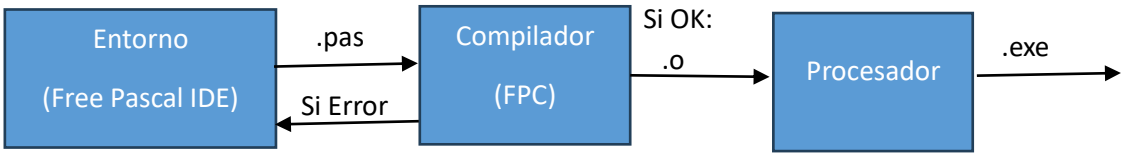
Asentemos a continuación nuestras bases. ¿Cómo podemos hacer un programa totalmente funcional y que lo comprenda el ordenador?

En primer lugar, debemos conocer en qué ambientes puede un ordenador ser capaz de reconocer un programa escrito en un lenguaje y procesarlo. Hablamos de los **compiladores** y los **entornos de programación**.

Los entornos son **aplicaciones idóneas donde podremos crear archivos con una extensión la cual el compilador puede leer su contenido** (que es nuestro programa codificado en ese determinado lenguaje de programación) **y traducir las instrucciones de dicho programa al lenguaje máquina** para que el procesador de nuestro ordenador ejecute la tarea encomendada.

Cabe diferenciar entre entorno de programación, que es donde se **crea el archivo de nuestro programa** y se envía al **compilador**; y este último, que es lo que **lee nuestro archivo** y **transforma** el conjunto de instrucciones de nuestro lenguaje de programación **al lenguaje máquina** y lo envía al procesador. Muchas veces, los compiladores no se incluyen en la instalación del entorno, y deben instalarse aparte. Por ejemplo, para ADA debemos instalarnos el compilador GNAT y el entorno AdaGide, y para Python y C debemos descargarlos dentro de la propia aplicación de VS Code.

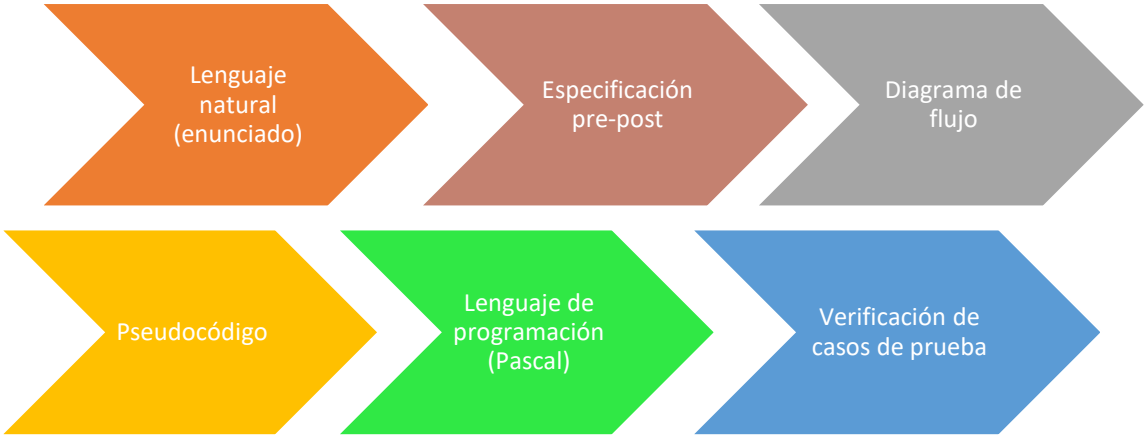
Nosotros usaremos a partir de ahora por defecto el lenguaje Pascal. Para programar en este lenguaje, basta con descargar Free Pascal, que incluye ambos, entorno de programación (Free Pascal IDE) y compilador (FPC).



En este entorno, codificaremos nuestros programas en el lenguaje Pascal, y los guardaremos en la extensión propia del lenguaje, pas. Además, este entorno permite ejecutar nuestros programas. Para ello, el entorno envía nuestro archivo pas al compilador. Si todo es correcto, el compilador crea nuestro código máquina de extensión o y lo envía al procesador, que ejecutará el programa, creando un ejecutable (.exe). En caso contrario, el compilador enviará un mensaje de error y la línea del documento en la que se ha detectado.

A modo de anexo, se adjuntarán guías breves para programar en los cuatro lenguajes que usaremos en este curso: Pascal (por defecto), ADA, C y Python.

Vamos a diseñar nuestro primer sencillo programa, será un programa en el que dado el radio de un círculo, calcule el área de este. Su fórmula es: $A = \pi r^2$, siendo el valor de π (pi) aproximadamente 3.1416. Como principiantes en la programación, se sugiere seguir la siguiente serie de **pasos para** poder **diseñar cualquier** tipo de **programa**:



1. **Leer y comprender el enunciado**, qué se pide, qué datos de entrada tenemos y qué datos de salida se esperan. Detectar también posibles ambigüedades.
2. Realizar la **especificación pre-post** del programa, los tipos de datos esperados, qué condiciones se esperan que cumplan los datos de entrada y los datos de salida.
3. Diseñar un buen **diagrama de flujo** ayuda a visualizar mejor las **ideas principales** del algoritmo. Establece la serie de pasos que hay que realizar para ejecutar el problema en un borrador u hoja en sucio y después expresa esos pasos en bloques y sus transiciones en líneas de flujo.

4. Observando las **estructuras de control** que contiene el diagrama de flujo, podemos fácilmente intuir el **algoritmo en pseudocódigo**. En el pseudocódigo, podemos utilizar todas las herramientas que se nos ocurran, independientemente de si estas son soportadas por el lenguaje solicitado o no.
5. Finalmente, **desarrollamos el programa** cumplimentando con la estructura similar al texto instructivo que hemos visto: cabecera del título, especificación, declaraciones e **implementación del algoritmo en ese lenguaje de programación**. En este paso, nuestras herramientas son limitadas y **debemos ajustarnos a las reglas gramaticales** del lenguaje.
6. No viene mal agrupar en una tabla una serie de **casos de prueba significativos** y sus resultados esperados y **ejecutar el** nuevo **programa** confiriendo esos valores a los datos de entrada para comprobar que **se obtienen los resultados esperados**.

Esto es sólo una **metodología básica e inicial** que se sugiere usar con el fin de ayudar al joven programador a diseñar sus programas correctamente, especialmente aquellos más complejos. En cuanto avancemos más, podremos irnos más directos al grano, al paso 5, pero al menos en esta parte nuclear de la asignatura es altamente recomendable seguir estos pasos, sobre todo si eres nuevo en el mundo de la programación.

Recordamos que un programa es correcto cuando **coincide lo que se obtiene de valor salida al ejecutar el programa y lo que se debería obtener según la especificación** del programa, por ello es muy importante hacer hincapié en realizar buenas especificaciones y en procurar seguir el diseño del programa según ese contrato. Un programa que no sigue su especificación no es correcto.

Además, recuerda que todos los caminos llevan a Roma, por lo que **dos programas distintos pueden resolver el mismo problema correctamente**, y por tanto los dos son iguales de válidos. Su diferencia radica simplemente en la sencillez con la que se puede seguir el diseño del programa (longitud de las instrucciones, complejidad de las estructuras de control, nº de variables a declarar, etc.) o eficiencia. Entre todas las características de los programas, **Bases de la Programación priorizará la corrección**.

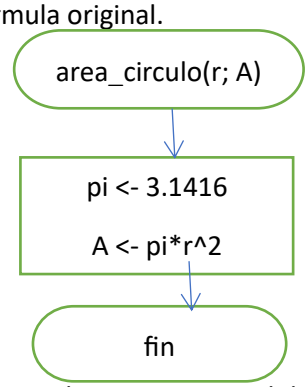
A continuación, procedemos a diseñar el programa propuesto del área del círculo.

El área de círculo, que es lo que queremos obtener, recibe como única incógnita el radio del círculo, único dato de entrada. El enunciado no es ambiguo, pues aunque no se ofrece la unidad de medida del radio, la unidad que el usuario decidiera tomar no afectaría en absoluto al resultado del área.

El tipo de dato radio puede ser o bien entero o bien real, pero asumiremos que r es real porque es el tipo de dato menos restringido de ambos. La especificación pre-post del programa es la siguiente:

- Datos de entrada: un real r, radio de la circunferencia
- Precondición: r > 0.0
- Datos de salida: el área del círculo, A, nº real
- Postcondición: A = 3.1416 * r^2

Cabe recalcar que **si no se menciona nada sobre entrada por teclado o salida en pantalla, los datos de entrada y salida se obtienen desde memoria**, y será así hasta dar el tema 7. Procedemos a continuación a diseñar su respectivo diagrama de flujo y posteriormente, el pseudocódigo. En este caso se trata de un algoritmo muy sencillo, ya que tan sólo hay que aplicar una fórmula matemática. No obstante, para optimizar mejor el código, vamos a declarar una variable PI <- 3.1416, siendo más fieles a la fórmula original.




```
area_circulo(r; A){
  pi := 3.1416;
  A := pi * r^2;
}
{ En pseudocódigo, deben indicarse los datos
de E/S estándar o de memoria que se toman }
```

En este caso observamos que el diagrama de flujo sigue una **estructura secuencial**, lo que quiere decir que **ejecuta las instrucciones una a una, en orden y obligatoriamente**. Vamos a ver a continuación cómo desarrollar un programa en estructura secuencial.

En primer lugar, dos cosas comunes a todos los programas son el **título del programa y la especificación** de este. El título debe tener un nombre que al leerlo nos dé una idea de para qué sirve el programa. La especificación preferiblemente debe ser la pre-post, y debe introducirse a modo de comentario. Ambas partes del programa dependen por supuesto del lenguaje de programación que utilicemos.

En Pascal, la cabecera del programa debe ir precedida de la palabra reservada program, y sucedida por un punto y coma (;) y los comentarios se introducen entre dos llaves. El inicio de nuestro programa en Pascal debería ser el siguiente:

```
program area_circulo;

{ Datos de entrada: un real r, radio de la circunferencia
Precondición: r >0.0
Datos de salida: el área del círculo, A, nº real
Postcondición: r = 3.1416 * r^2 }
```

Posteriormente, debemos **declarar toda variable o tipo no primitivo** que utilicemos, lo que nos abre una nueva sección en este tema. Estas declaraciones actúan como si fueran una lista del **inventario** que hay que tener a mano para seguir una receta o una tarea.

4.1 Declaraciones

En los lenguajes de programación **de alto nivel**, cuando se quiera utilizar una variable, **es obligatorio indicar anteriormente el tipo de dato que se va a usar con ella**, como si fuera un listado de las cosas que hay que tener antes de ejecutar una tarea. Como se ha observado **en el pseudocódigo**, a diferencia de en el programa, **no es obligatorio declarar variables**.

La manera en la que se realizan las declaraciones vuelve a depender por supuesto del lenguaje de programación. Además, cabe distinguir entre constantes y variables.

Las constantes se declaran al principio del programa y, como su nombre indica, tienen la peculiaridad de que, **una vez declarado su valor, lo mantienen durante toda la ejecución** del algoritmo. Si el algoritmo intentara por lo que fuera manipular el valor de una constante (p.ej. se intenta ejecutar la instrucción pi:=2*pi, siendo pi=3.1416 declarada como constante), daría lugar a un error de compilación; lo que le confiere mayor seguridad a la hora de programar. **La inicialización del valor de una constante es obligatoria**.

Por otro lado, **las variables pueden cambiar de valor a lo largo de la ejecución del código**, de ahí su nombre. Su inicialización no es obligatoria, pero si se ejecutara una instrucción en donde se necesitara su valor preciso y esta no está o bien inicializada o bien asignada (p.ej. declarar una variable x y se ejecuta la instrucción y := x + 1 sin conferir anteriormente un valor a x), se producirá un error de compilación.

Evidentemente, también dará un error de compilación si ejecutamos una instrucción que utiliza una variable no declarada.

Por lo que, siguiendo las reglas de Pascal, las declaraciones se realizan antes del algoritmo, y las constantes y las variables se declaran separadas en dos secciones, una sucedida por la palabra reservada const y otra por la palabra var, siguiendo la siguiente sintaxis:

<declaración_constantes> ::= <nombre_constante> : <tipo_dato> := <valor_constante> ;

<declaración_variables> ::= <nombre_variable> : <tipo_dato> [:= <valor_inicio>] ;

```
const
   PI : Real := 3.1416;
var
   A, r: Real;
```

Cabe destacar que **por convenio, las constantes se escriben en mayúsculas mientras que las variables se escriben en minúsculas**. Este convenio se establece porque en lenguajes de bajo nivel, como son débilmente tipados y por ende no se realizan declaraciones, los desarrolladores distinguen así entre lo que está diseñado para que sea constante y lo que está diseñado para que sea variable. Sin embargo, los lenguajes de bajo nivel como Python no logran evitar que a lo largo del programa se pueda modificar el valor de una constante, ya que son tratadas como variables igualmente.

Por otra parte, otros lenguajes como C no distinguen entre la parte donde se escriben las declaraciones de la parte donde se escriben las asignaciones y resto de instrucciones.

Aparte de constantes y variables, todo tipo de dato no primitivo del lenguaje también debe declararse. Dependiendo del tipo de dato, la manera de declararlos también variará, pues no es lo mismo una enumeración que una rodaja, por ejemplo.

En Pascal se procedería siguiendo esta sintaxis, sucedida de la palabra reservada type:

<declaración_enumeración> ::= <nombre_tipo> = (<tag1>, … , <tagn>);

<declaración_rodaja> ::= <nombre_tipo> = <valor_inicio>..<valor_fin>;

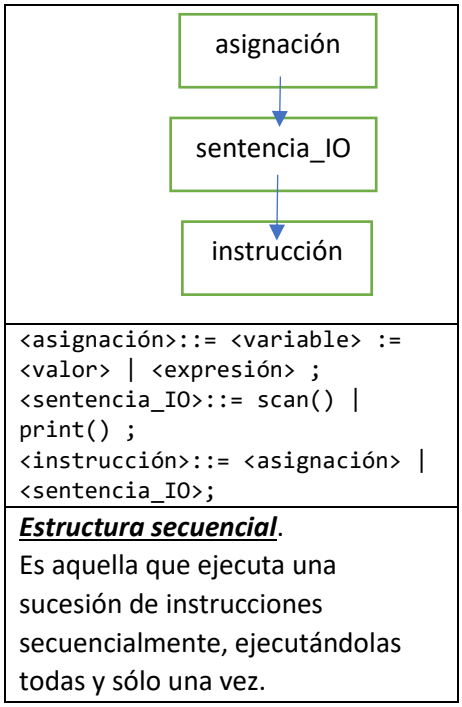
Así por ejemplo, para definir en un programa de Pascal la enumeración de días de la semana y la rodaja de nº positivos, se procedería de la siguiente manera:

```
type
   dias = (L, M, X, J, V, S, D);
   natural = 0..MaxInt;
```

Como siempre, esto depende bastante del lenguaje de programación, y de hecho, en C no se aceptan rodajas y en Python no se aceptan ninguno de los dos TDDs vistos por ahora.

4.2 Estructura secuencial

La **estructura secuencial** se refiere a un diseño del algoritmo en el cual **una serie de instrucciones son ejecutadas en el orden en el que aparecen**, de arriba a abajo.




```
<asignación>::= <variable> :=
<valor> | <expresión> ;
<sentencia_IO>::= scan() |
print() ;
<instrucción>::= <asignación> |
<sentencia_IO>;
```

Estructura secuencial.
Es aquella que ejecuta una sucesión de instrucciones secuencialmente, ejecutándolas todas y sólo una vez.

La estructura secuencial tiene las siguientes características:

- Sus instrucciones se siguen de forma que **el valor de salida que se obtiene tras ejecutar una instrucción es utilizado como valor de entrada para ejecutar su siguiente** instrucción
- El programa **sólo** tiene **un punto de entrada y sólo un punto de salida**
- La **ejecución** de las instrucciones es **unidireccional** (de arriba abajo, sin retroceder nunca), **lineal** (una a una y en orden) **y de una única vez**.
- La lectura de las instrucciones es sencilla, y por lo tanto **fácil de mantener**, ya que sólo se emplean **asignaciones y operaciones básicas** (entre ellas operaciones características de cada tipo de dato y operaciones de E/S que veremos en el tema 7)

En el caso del programa del área del círculo, observamos que se trata de un programa que se puede resolver mediante únicamente asignaciones, que actúa como función matemática (y por lo tanto, un único punto de entrada y un único punto de salida), que sólo es necesario ejecutar una sola vez un número de instrucciones, que tienen un orden concreto y que además, el resultado de la primera determina el resultado de la segunda. Por todo ello, podemos inferir que la estructura más adecuada para diseñar el algoritmo de este ejemplo es la estructura secuencial.

Siguiendo la sintaxis de Pascal de las asignaciones, que es la siguiente:

```
<asignación>::= <variable> := <valor> | <expresión> ;
```

Tendríamos únicamente una instrucción a ejecutar, ya que la instrucción de pseudocódigo en donde se había asignado a pi el valor 3.1416, se sustituye por la declaración de la constante. Esta instrucción corresponde con la ejecución de la fórmula matemática $A = \pi r^2$

```
A := PI*r*r;
```

Hay que tener en cuenta que la operación exponenciación (como la que se presenta, r^2), no existe en Pascal, por lo que debemos pensar en otras formas de ejecutar dicha operación. Sabemos que por definición: $x^n = x \cdot x \cdot x \cdot x \ldots$ n veces, luego, $x^2 = x \cdot x$.

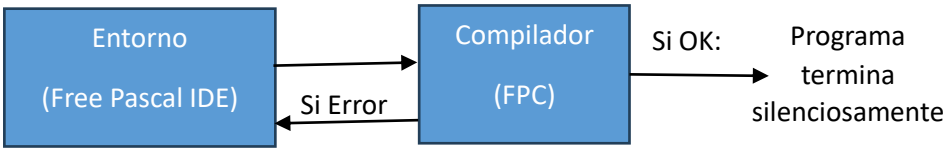
La secuencia de instrucciones de nuestro programa, en Pascal, debe ir comprendida entre las palabras reservadas begin y end. El programa en Pascal quedaría finalmente de la siguiente forma:

```
program area_circulo;
{$Assertions ON}
{ Datos de entrada: un real r, radio de la circunferencia
Precondición: r >0.0
Datos de salida: el área del círculo, A, nº real
Postcondición: r = PI * r^2, con PI = 3.1416}

const
   PI : Real := 3.1416;
var
   A, r: Real;
begin
   r := 1; {Inicializar datos de entrada}

   A := PI*r*r;

   Assert(A = PI); {Verificar caso de prueba}

end.
```

No obstante, a la hora de programar nuestros primeros programas, debemos tener en cuenta tres aspectos:

- Estos programas **no reciben entrada por teclado ni imprimen en pantalla**
- Por lo dicho anteriormente, esto causará un error de compilación porque los datos de entrada que no han sido inicializados tienen que procesarse en una asignación posterior. Por ello, es **obligatorio inicializar en el código los datos de entrada**, preferiblemente con los valores de un **caso de prueba** significativo
- No estaría de más **comprobar el correcto funcionamiento del programa**. Para ello, utilizamos la funcionalidad **Assert**. Para habilitarla, se ha de escribir tras la cabecera del programa `{$Assertions ON}`. La sintaxis de esta funcionalidad es: `<aserción>::= Assert( <dato_de_salida> = <valor_esperado> )` y con esto, **comprobamos que dados los datos de entrada del caso de prueba, obtengamos los datos de salida esperados**.

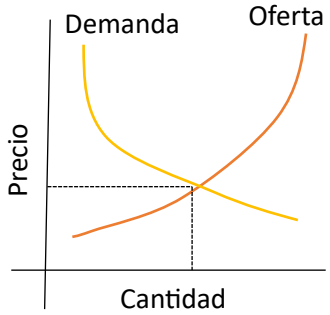
Si la aserción es verdadera, el programa termina silenciosamente, ya que ni recibe ni imprime datos en pantalla (no hay comunicación alguna con el usuario), lo que significa que **el programa es parcialmente correcto y** que **el caso de prueba se ha verificado** correctamente. **En caso contrario, el compilador lanzará un error de aserción** el cual será visible desde nuestro entorno de programación, por lo que **el programa no es correcto**.



A continuación, expondremos otro programa más complicado en el cual la serie de instrucciones a ejecutar ya no es tan obvia. Además, codificaremos este programa en ADA, C y Python para que se puedan observar las principales diferencias entre estos lenguajes y Pascal.

"Ecn. El principio de Economía que se utiliza en el ámbito empresarial para que las compañías que ofrezcan sus productos o servicios obtengan el máximo beneficio posible establece que lo ideal para el vendedor sería tener ventas altas y precios altos, pero esto contrasta con lo que el comprador desea, que son precios bajos. El principio que busca solución a este conflicto es la ley de la oferta y la demanda.

Si observamos el modelo de telaraña (gráfico a la derecha), se puede ver en un espacio donde sus variables son el precio del producto y la cantidad de productos existente dos funciones características, la de la oferta del vendedor (que aumenta el precio según la cantidad de productos a vender) y la de la demanda del comprador (que disminuye el precio según el mismo criterio)



El objetivo del ejercicio es encontrar el punto donde intersecan ambas funciones, el umbral de rentabilidad . Siendo la ecuación de la oferta $X_O = Q_O P - C_f$ y la ecuación de la demanda $X_D = Q_{max} - Q_D P$, siendo X_O, las unidades ofertadas; X_D, las unidades demandadas; C_f, los costes fijos de producción; Q_{max}, el stock máximo del producto; P, el precio de la unidad; y la pendiente de las rectas Q_O, Q_D.

Diseña un programa en el que dados los costes fijos de producción, el stock máximo del producto y las pendientes de ambas rectas, calcule el precio y la cantidad del producto necesarios para que estos valores cumplan con el umbral de rentabilidad"

En resumen, se pide el corte entre dos rectas de las que se saben sus ecuaciones y algunos parámetros, entre ellos C_f, Q_max y Q_O, Q_D; el corte entre ambas rectas es el punto (X, P), siendo X la cantidad y P el precio.

Si las rectas intersecan, los valores de X y P se igualan, por lo que X_O = X_D, por lo que esta igualdad $Q_O P - C_f = Q_{max} - Q_D P$, se cumple. De esta sencilla ecuación es donde despejaremos el valor de P, y una vez obtenido el precio de equilibrio, sustituimos en una de las dos rectas la incógnita P para obtener la cantidad de equilibrio X.

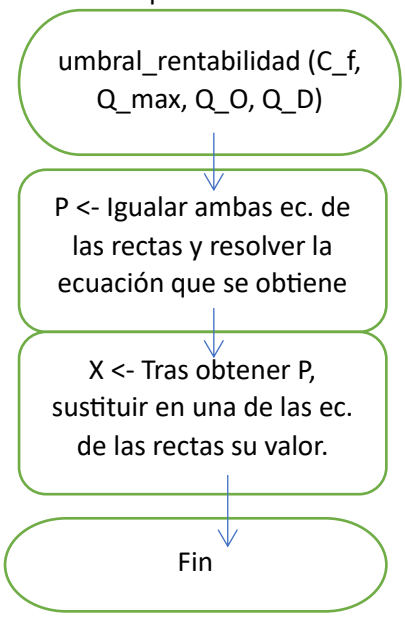
La solución que se obtiene al despejar P es $P = \frac{Q_{max}+C_f}{Q_O+Q_D}$. Tras obtener este valor, obtendremos X sustituyendo P por su valor p.ej. en la recta de la oferta.

La solución del problema en pseudocódigo sería la siguiente:

```
umbral_rentabilidad (C_f, Q_max, Q_O, Q_D; P, X){
      P := (Q_max + C_f) / (Q_O + Q_D);
      X := Q_O*P – C_f;
}
```

Luego, la especificación pre-post de este programa es la siguiente. Para simplificar el programa, asumiremos que todos los parámetros del programa son reales, aunque parámetros como cantidad de productos puedan optimizarse mejor como enteros:

- Datos de entrada: cuatro reales, C_f, Q_max, Q_O, Q_D; costes fijos, stock máximo y pendientes de las rectas de oferta y demanda.



- Precondición: C_f, Q_max, Q_O, Q_D >= 0.0
- Datos de salida: dos reales, P, X, precio y cantidad del umbral de rentabilidad
- Postcondición: P = (Q_max + C_f) / (Q_O + Q_D) ^ X = Q_O*P – C_f

Ya se ha resuelto prácticamente la mitad del problema, una vez se haya leído el enunciado e identificado lo que se pide, lo que se tiene, lo que se debe obtener y la serie de pasos a ejecutar, el último paso que se ejecuta es la implementación del algoritmo en el lenguaje de programación elegido, y verificar que el programa funciona correctamente.

Se muestra a continuación cómo se vería dicho programa en los siguientes lenguajes de programación:

1. **PASCAL**

```
program umbral_rentabilidad;
{$Assertions ON}
type
      PosReal = 0.0..MaxReal;
var
      C_f, Q_max, Q_O, Q_D, P, X: PosReal;
begin
      C_f := 500.0; Q_max := 700.0; Q_O := 3.0; Q_D := 1.0;
      P := (Q_max + C_f) / (Q_O + Q_D);
      X := Q_O*P – C_f;
      Assert(P = 300.0 and X = 400.0);
end.
```

2. ADA

```
with Ada.Assertions; use Ada.Assertions;
subtype PosFloat is Float range 0.0..Float'Last;
procedure umbral_rentabilidad (C_f, Q_max, Q_O, Q_D: in PosFloat;
                               P, X: out PosFloat) is
begin
      C_f := 500.0; Q_max := 700.0; Q_O := 3.0; Q_D := 1.0;
      P := (Q_max + C_f) / (Q_O + Q_D);
      X := Q_O*P – C_f;
      pragma Assert(P = 300.0 and X = 400.0);
end umbral_rentabilidad;
```

3. Python

```
# umbral_rentabilidad
C_f, Q_max, Q_O, Q_D = 500.0, 700.0, 3.0, 1.0
P = (Q_max + C_f) / (Q_O + Q_D)
X = Q_O*P – C_f
assert(P == 300.0 and X == 400.0)
```

4. C

```
#include <assert.h>
void main() { /* umbral_rentabilidad */
        double C_f, Q_max, Q_O, Q_D, P, X;
        C_f = 500.0; Q_max = 700.0; Q_O = 3.0; Q_D = 1.0;
        P = (Q_max + C_f) / (Q_O + Q_D);
        X = Q_O * P - C_f;
        assert(P == 300.0 && X == 400.0);
}
```

Las normas gramaticales que se han seguido para escribir el programa en cada lenguaje de programación se encuentran en el anexo relacionado a este tema. **Hay que tener en cuenta a la hora de programar las limitaciones de cada lenguaje y adaptar el pseudocódigo a estas**.

Por ejemplo, fijémonos en los dos primeros programas. En ambos programas se ha establecido un nuevo TDD rodaja denominado PosReal o PosFloat, el cual reúne a todos los números reales mayores o iguales que 0.0. Esto restringe a nivel computacional el rango de valores que pueden tener los parámetros reales que les pasemos al programa. Esto quiere decir que si pasamos un número real negativo como valor de uno de los parámetros, el programa dará lugar a un error de tipado. Es por ello que la precondición de nuestro programa evoluciona a true, ya que no hay ningún número real positivo que incumpla la propiedad C_f, Q_max, Q_O, Q_D >= 0.0. Sin embargo, si pasamos el mismo real negativo a los parámetros de los programas codificados en C o en Python, no habrá ningún tipo de error de ejecución y el programa calculará según ese valor negativo los valores de P y X; por lo que para que cumpla dicha propiedad, necesitamos precondicionar los valores de los parámetros reales.

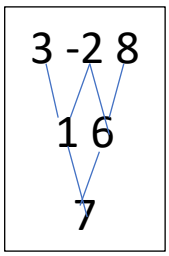
C es un lenguaje bastante peculiar también, pues no se distingue entre las partes donde se declaran variables de las que se asignan las instrucciones, y ambas se tratan como asignaciones por igual, salvo en la primera aparición de la variable, en donde sí es obligatorio por cada una de ellas declarar el tipo de dato correspondiente a la variable. Añadir la etiqueta del tipo de dato a una variable otra vez desde que ya se declaró por primera vez el tipo de dato, dará lugar a un error de compilación, sólo se declaran variables en su primera aparición.

Por último, Python es un lenguaje de bajo nivel, por lo que no es necesario declarar ninguno de los parámetros como reales u otro tipo de dato, directamente pasamos a las asignaciones. Esto quiere decir que si introducimos enteros, entonces el resultado puede salir o entero o real (p.ej. división no exacta entre dos enteros). Python tiene una manera muy especial de manejar los errores de tipado, y se verá en el tema 7, cuando veamos "Conversión de tipos"

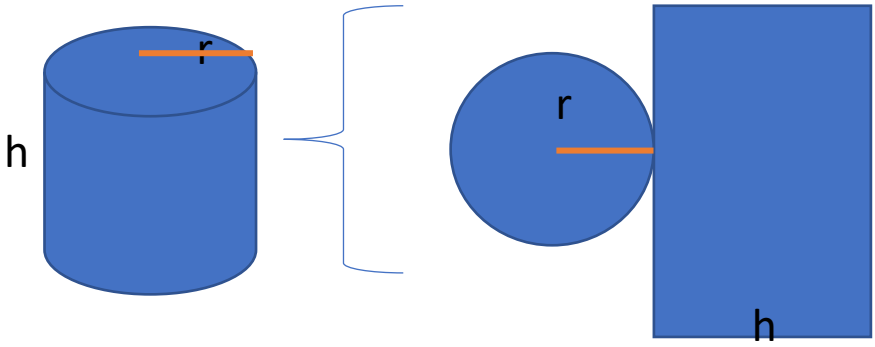
4.3 Ejercicios propuestos

Se sugieren realizar los siguientes ejercicios de programación. Estos ejercicios se deben realizar usando exclusivamente las herramientas que hemos visto hasta ahora, es decir, declaraciones y asignaciones. Se pueden hacer en el lenguaje de programación que uno crea conveniente, aunque preferiblemente usaremos Pascal; salvo que el enunciado indique que hay que escribir el programa en un lenguaje determinado. Los programas que denotan aplicaciones en el campo STEM se denotarán con la abreviatura de la rama STEM a la que pertenece.

65. (*) Mat. Diseña un programa en el que dados la pendiente de una recta m y su ordenada en el origen n, calcule el valor de y en el valor x del eje de abscisas. Recuerda que la ecuación de la recta es $y = mx + n$
66. (*) Diseña un programa en el que dadas diez notas medias de las diez asignaturas del curso, calcule la nota media del expediente académico en ese curso. Recuerda que para realizar la media se sumaban todos sus datos y se dividía dicha suma por el número de datos existente.
67. (*) Diseña un programa en el que dadas tres notas medias de un trimestre, calcula la nota media de la asignatura en ese año. Sin embargo, no todos los exámenes tienen el mismo peso, la evaluación del primer trimestre tiene un peso del 20% de la nota total, la del segundo trimestre, del 30% y la del tercero, el resto del peso.
68. (*) Diseña un programa en el que dados dos números enteros, indique si el valor de su suma devuelve un número par o si por el contrario, número impar
69. (*) Diseña un programa en el que dados dos números enteros, indique si el valor de su suma devuelve un número par o no pero esta vez sin realizar la suma
70. (*) Diseña un programa en el que dado un natural n, decide si este es o no un múltiplo satánico. n es múltiplo satánico cuando es múltiplo de 6 y de 66 o lo es de 666.
71. (*) En el videojuego "Brain Training", uno de sus ejercicios de agilidad era "Triángulo Aritmético" En este minijuego se mostraban tres números enteros, y su objetivo era hallar lo más rápido posible la suma de los dos pares que forman dicha secuencia. P.ej. para la secuencia 3, -2, 8, la suma del primer par es 3 + -2 = 1, la suma del segundo par es -2 + 8 = 6, por lo que la suma de los dos pares es 1 + 6 = 7. Diseña el programa que calcule el triángulo aritmético de tres números dados



72. (**) Mat. Diseña un programa que calcule dado el radio r y su altura h el volumen de un cilindro. Recuerda que el volumen de un cuerpo geométrico es $V = A_{base} * h$, siendo el área de su base el área de un círculo ($A = \pi r^2$), pi lo aproximaremos a 3.1416
73. (**) Mat. Diseña un programa que calcule el área de un tanque de agua cilíndrico dado su radio r y su altura h. La base de arriba del cilindro queda descubierta, ya que desde allí se introduce el agua. Luego, necesitamos construir la base de abajo y la superficie cilíndrica que encierra el agua. Recuerda que la longitud de la circunferencia es $2\pi r$



74. (**) Diseña un programa en el que dados tres números reales a, b, c, devuelva para cada una de esas variables cuál es el porcentaje que supone entre el total que forman la suma de ellas tres.
75. (**) Diseña un programa en el que dados dos números enteros a y b, intercambie sus valores. P.ej. si se pasan los valores 3, 8; se deben devolver los valores 8, 3.
76. (**) Dado un número entero de cuatro cifras, diseña un programa que sea capaz de descomponerlo en sus cuatro dígitos, almacenando cada uno en una variable del tipo entero.
77. (**) Dado un carácter, diseña un programa que indique si este es un dígito o no.
78. (**) Dado un carácter, diseña un programa que indique si este forma parte del alfabeto inglés o no. Así p.ej. la é, la ñ o la ü no entrarían en la definición.
79. (**) Dados dos caracteres x e y, siendo el primero mayúsculo y el segundo minúsculo, diseña un programa en el cual sus valores se intercambian. El programa debe respetar el hecho de que el carácter x sea siempre mayúsculo y el carácter y sea siempre minúsculo. P.ej. si se pasan los valores 'A', 'b'; se deben devolver los valores 'B', 'a'
80. (***) Dados dos dígitos a y b, diseña un programa que realiza la suma de sus dígitos. Esta se realiza sumando los valores numéricos de los dígitos y devolviendo el valor de dígito del resultado. P.ej. '3' + '2' = '5'. Pero a veces puede darse desborde, por ejemplo, en '6' + '9', sabemos que su suma da 15, pero devolverá su último dígito, '5', por lo que '6' + '9' = '5' El programa debe indicar también si hay desborde o no.
81. (***) Dadas dos letras minúsculas x e y, diseña un programa que indique si ambas letras forman un fonema trabado, es decir, si al juntar la letra x seguida de la letra y, se forman alguna de estas secuencias de consonantes: bl, br, cl, cr, dr, fl, fr, gl, gr, pl, pr, tr.
82. (***) Mat. Dados tres coeficientes a, b y c que forman parte de la ecuación cuadrática $ax^2 + bx + c = 0$, diseña un programa que devuelva sus dos soluciones reales. La operación raíz cuadrada debe implementarse aplicando el siguiente polinomio de aproximación, Δ (delta) denota el número del que se quiere calcular su raíz cuadrada:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$\sqrt{\Delta} = 1 + \frac{1}{2}(\Delta - 1) - \frac{1}{8}(\Delta - 1)^2$$

83. (***) Mat. Dados dos valores a, b del número complejo $a + bi$, diseña un programa que transforme la forma binómica del número en su forma polar $re^{i\theta}$, obteniendo los valores r y arg(θ). Recuerda que $r = \sqrt{a^2 + b^2}$ y que arg = $\arctan(\frac{b}{a})$, ángulo expresado en radianes y comprendido entre 0 y 2π. El polinomio de aproximación de la función arco tangente es:

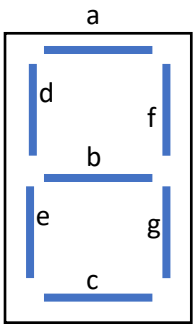
$$\arctan(\theta) = \theta + \frac{\theta^3}{3}$$

84. (***) Fis. Diseña un programa que calcule el tiempo en segundos necesario para que un vehículo que vaya a v m/s pueda incorporarse en una autovía y alcanzar los vf m/s en el final de los x metros del carril de aceleración. v denota la velocidad inicial del vehículo al entrar al carril de aceleración y vf, la velocidad máxima de la autovía. Las ecuaciones del MRUA son:

$$MRUA\begin{cases} v = v_o + at \\ x = x_o + v_o t + \frac{1}{2}at^2 \\ v^2 = {v_o}^2 + 2ax \end{cases}$$

85. (***) Fis. Cuando un vehículo circula por una cuesta abajo, existe la posibilidad de que los frenos fallen, por ello, se diseñan apartaderos de frenado de emergencia. Diseña un programa en el que dada la velocidad máxima de una vía v en m/s, calcule la longitud necesaria para un apartadero diseñado para que el vehículo sea inmovilizado en t segundos. El vehículo entra a una velocidad máxima de un 30% superior a la dada

86. (****) Diseña un programa en el que dadas dos horas h1:min1:sg1 y h2:min2:sg2, calcule cuánto tiempo ha transcurrido desde la hora h1:min1:sg1 hasta la hora h2:min2:sg2, manteniendo en el resultado el formato de salida en horas, minutos y segundos. Estrategia: calcula para ambas horas cuántos segundos han transcurrido desde las 00:00 y luego convierte el resultado de su diferencia en horas, minutos y segundos.
87. (****) Repite el mismo programa que el anterior pero restando directamente ambas horas en su respectivo sistema sexagesimal
88. (****) Inf. Este es un problema muy típico de la asignatura Diseño de Sistemas Digitales, que se imparte en el grado de Ingeniería Informática. El objetivo de este problema consiste en diseñar un decodificador de 7 segmentos, también llamado BCD (Binary Coded Decimal)



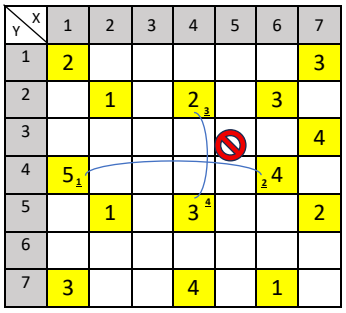
Este decodificador lo que hace es interpretar una serie de señales binarias, que corresponden al valor numérico en base 2 del dígito solicitado, y tras pasar dichos valores al decodificador, se devuelve como resultado una serie de siete señales binarias en forma de segmentos.

Aquí a la izquierda se muestra la idea de cómo se debería ver el número en pantalla. Este formato de número se suele encontrar en antiguas máquinas expendedoras o en termómetros públicos.

Así por ejemplo, si queremos imprimir en la pantalla el número 0, se encenderán todos los segmentos excepto el b, y para imprimir el 1, se encenderán únicamente los segmentos f, g.

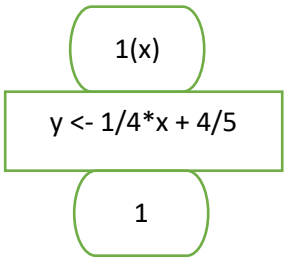
Diseña un programa en el que dado un dígito, se configuran las expresiones lógicas por las que cada uno de los siete segmentos se enciende. Se pide también que la resolución del problema sea lo más efectiva posible, es decir si sabemos que un segmento se enciende o no para ciertos valores y estos se pueden agrupar en una característica común, expresarlo como subconjunto.

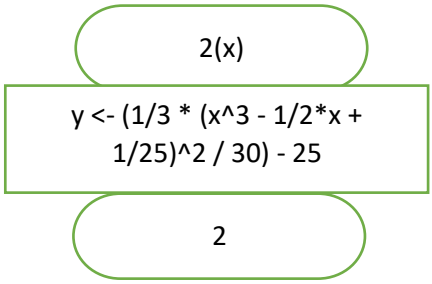
89. (****) Dados dos puentes, uno de extremos x1, y1; x2, y2 y otro de extremos x3, y3; x4, y4. (xi, yi simbolizan las coordenadas de la isla i del tablero Hashi), determina si dichos puentes pueden colocarse en el tablero simultáneamente sin superponerse uno sobre el otro. Los puentes dados descartan cualquier escenario en el que un puente pase por encima de una isla. Además, solo pueden disponerse de manera vertical (de arriba abajo) u horizontal (de izquierda derecha, en esos órdenes).

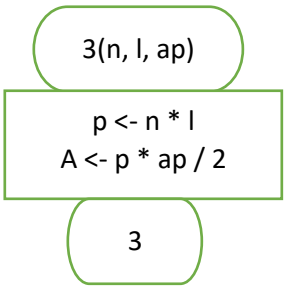


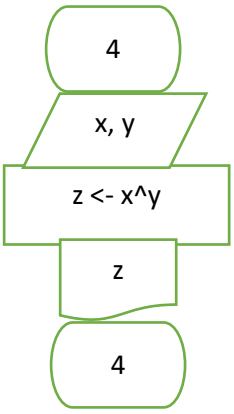
90. (*****).Diseña un programa en el que dados una hora completa en horas, minutos y segundos, un día, un mes y un año, devuelva su hora epoch. La hora epoch se define como el tiempo transcurrido en segundos entre el 1 de enero de 1970 a las 00:00 y la fecha dada. Recuerda que no todos los meses tienen la misma cantidad de días: enero, marzo, mayo, julio, agosto, octubre y diciembre tienen 31 días; abril, junio, septiembre y noviembre tienen 30 días; y febrero tiene 28 días (29 días si es año bisiesto) Un año es bisiesto si este es múltiplo de 4 pero no un año secular (aquel que termina por 00) o bien este es múltiplo de 400

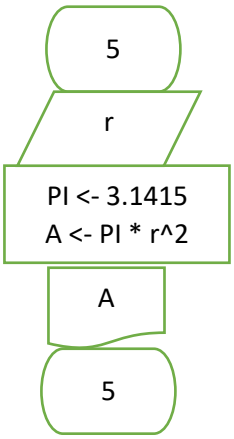
SOLUCIONARIO

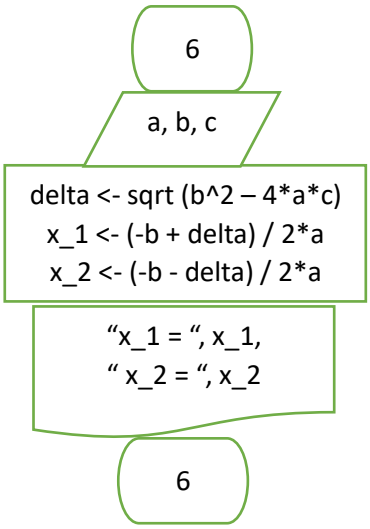


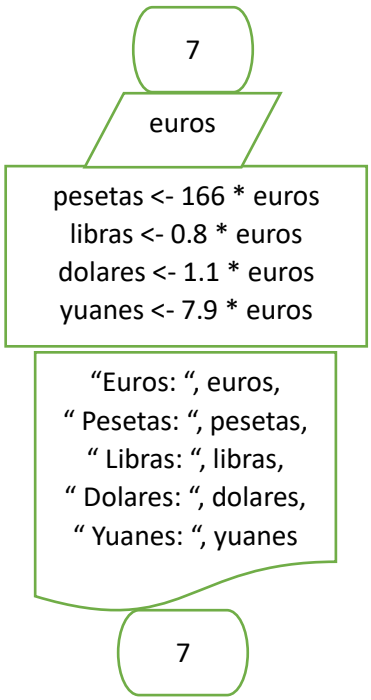


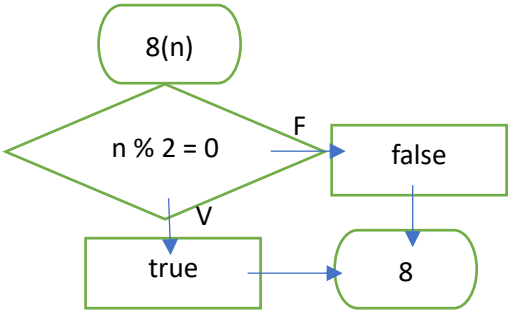


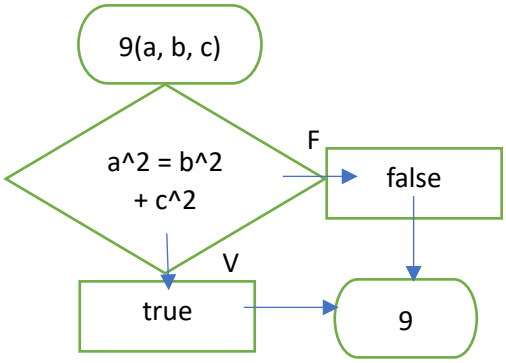


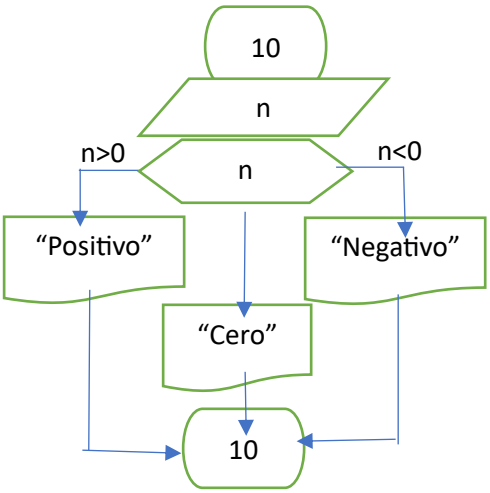


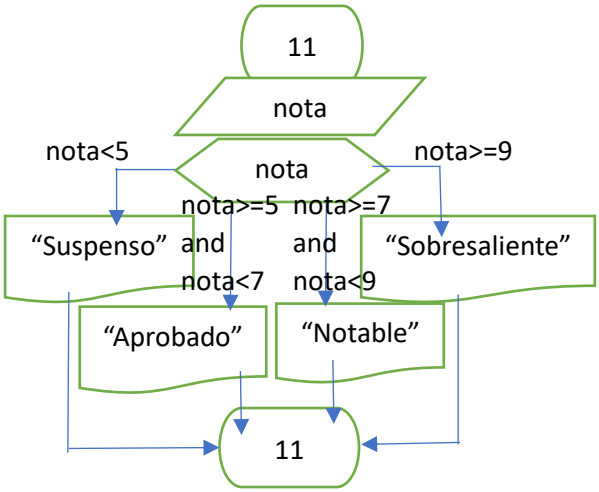


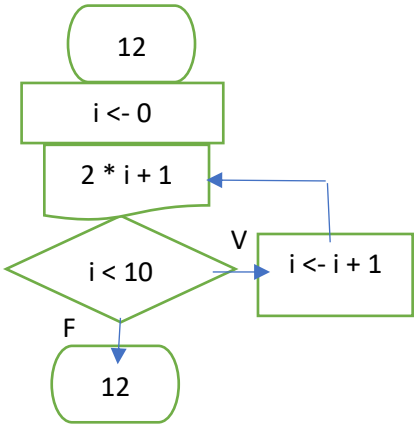


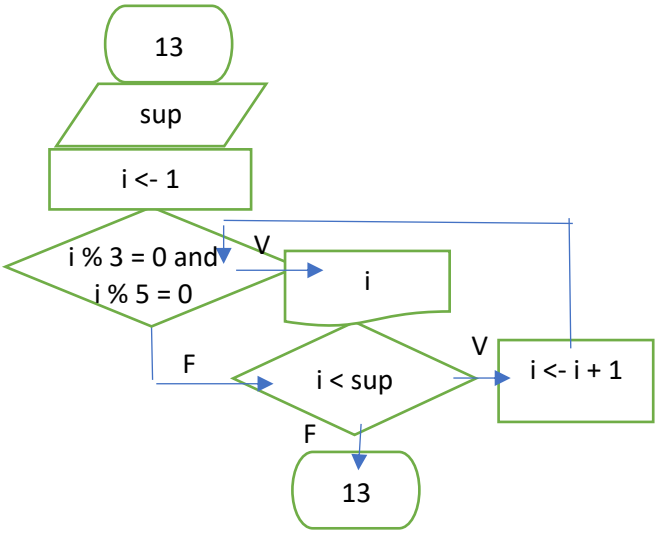


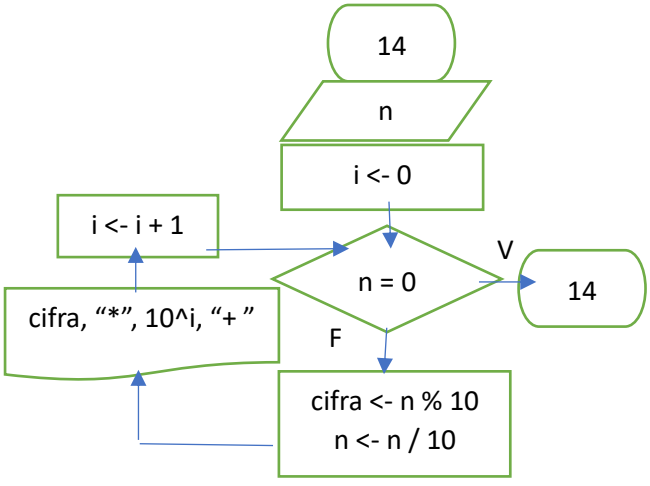


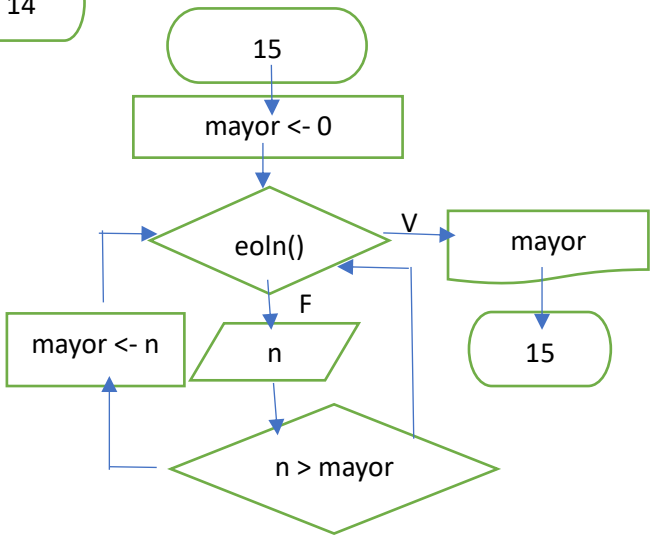

10
n
n>0
n
n<0
“Positivo”
“Negativo”
“Cero”
10

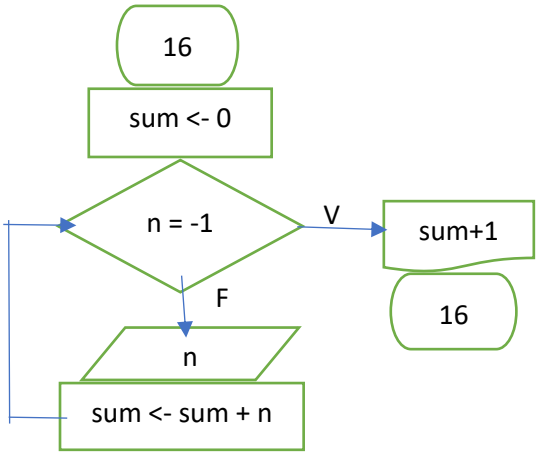

11
nota
nota<5
nota
nota>=9
nota>=5 and nota<7
nota>=7 and nota<9
“Suspenso”
“Sobresaliente”
“Aprobado”
“Notable”
11

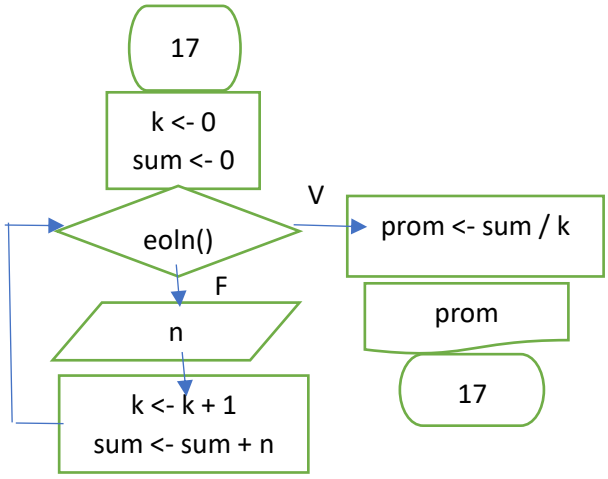

12
i <- 0
2 * i + 1
i < 10
V
i <- i + 1
F
12

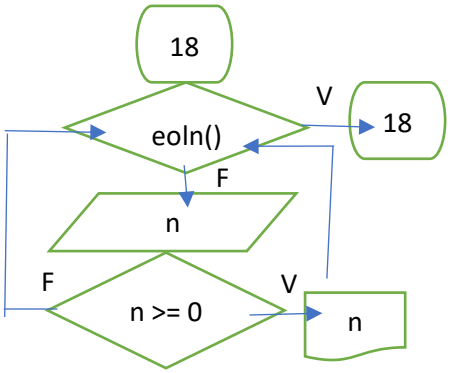

13
sup
i <- 1
i % 3 = 0 and i % 5 = 0
V
i
F
i < sup
V
i <- i + 1
F
13


14
n
i <- 0
i <- i + 1
n = 0
V
14
cifra, “*”, 10^i, “+ ”
F
cifra <- n % 10
n <- n / 10


15
mayor <- 0
eoln()
V
mayor
F
mayor <- n
n
15
n > mayor








```
algoritmo_19{
      scan(a, b ,c);
      delta := sqrt (b^2 – 4*a*c);
      x_1 := (-b + delta) / 2*a;
      x_2 := (-b - delta) / 2*a;
      print(“x_1 = “, x_1,
      “ x_2 = “, x_2);
}
```

```
algoritmo_20{
       scan(euros);
       pesetas := 166 * euros;
       libras := 0.8 * euros;
       dolares := 1.1 * euros;
       yuanes :=  7.9 * euros;
       print(“Euros: ”, euros);
       print(“ Pesetas:”, pesetas);
       print(“ Libras: ”, libras);
       print(“ Dolares: ”, dolares);
       print(“ Yuanes: ”, yuanes);
}
```

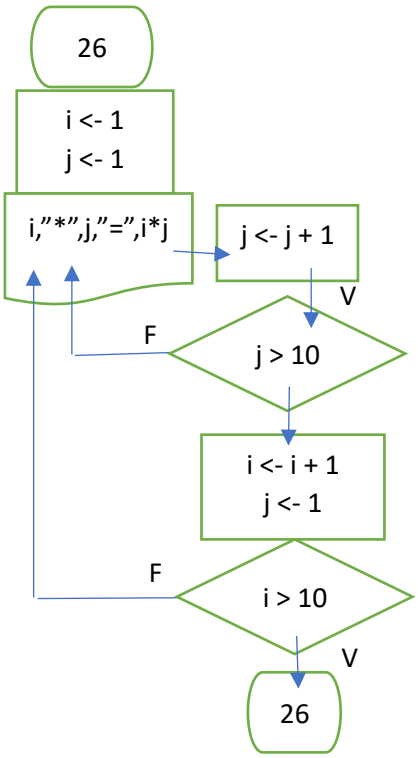
```
algoritmo_21{
      scan(nota);
      if (nota<5)
            print(“Suspenso”);
      else if (nota>=5 and nota<7)
            print(“Aprobado”);
      else if (nota>=7 and nota<9)
            print(“Notable”);
      else if (nota>=9)
            print(“Sobresaliente”);
}
```

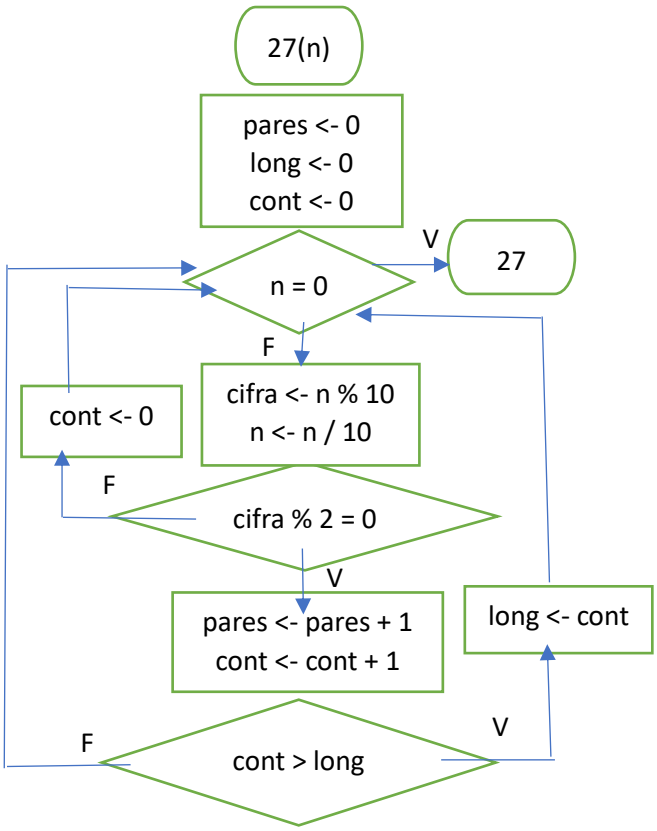
```
algoritmo_22{
   for i in range 0 to 9 do
       print (2*i+1);
}
```

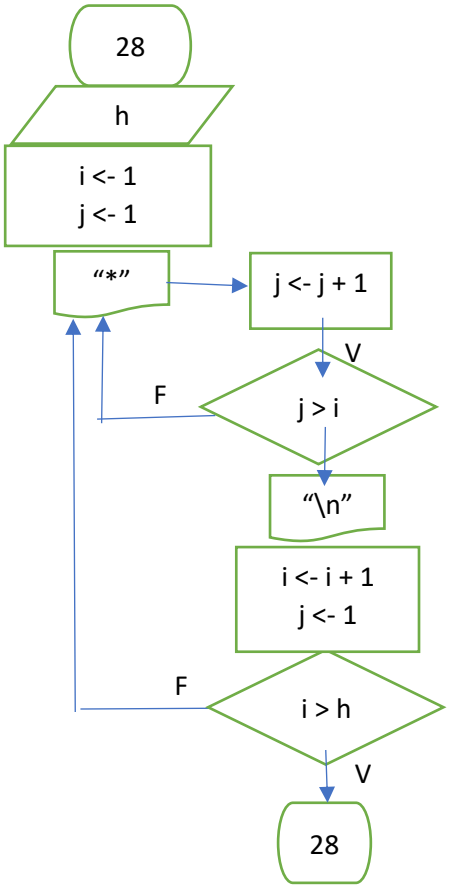
```
algoritmo_23{
       scan(sup);
       for i in range 1 to sup do
             if (i mod 3 = 0
             and i mod 5 = 0) then
                   print(i);
}
```

```
algoritmo_24{
       scan(n);
       i := 0;
       repeat
             cifra := n % 10;
             n := n / 10;
             print(cifra, “*”,
             10^i, “+ ”);
             i := i + 1;
       until (n=0);
}
```

```
algoritmo_25{
	sum := 0;
	repeat
		scan(n);
		sum := sum + n;
	until (n=-1);
	print(sum+1);
}
```







29. El enunciado de este programa es ambiguo, estas son sus dos interpretaciones:

```
algoritmo_29(x, y){
        z := sqrt(x) – y;
}
algoritmo_29(x, y){
        z := sqrt(x – y);
}
```

30.
```
algoritmo_30(l){
        A := (l*sqrt(l^2– (l/2)^2))/2;
}
```

31. El enunciado de este programa es ambiguo, estas son sus dos interpretaciones:

```
algoritmo_31(x, y){
        z := x – y;
}
algoritmo_31(x, y){
        z := y – x;
}
```

32.
```
algoritmo_32(p){
cases of p:
        1: L;
        2: M;
        3: X;
        4: J;
        5: V;
        6: S;
        7: D;
}
```

33.
```
algoritmo_33(){
n := 0;
suma := 0;
repeat{
        scan(x);
        suma := suma + x;
        n := n + 1;
} until (eoln());
media := suma / n;
print(media);
}
```

34. algoritmo_34(){
    scan(día, mes);
    if ((día >= 21 and mes = 12) or (mes = 1) or (mes = 2) or (día < 21 and mes = 3)) {
        print("Invierno");
    } else if ((día >= 21 and mes = 3) or (mes = 4) or (mes = 5) or (día < 21 and mes = 6)) {
        print("Primavera");
    } else if ((día >= 21 and mes = 6) or (mes = 7) or (mes = 8) or (día < 21 and mes = 9)) {
        print("Verano");
    } else if ((día >= 21 and mes = 9) or (mes = 10) or (mes = 11) or (día < 21 and mes = 12)) {
        print("Otoño");
    }

35. El enunciado de este programa es ambiguo, estas son sus dos interpretaciones:
    algoritmo_35{
    print("10 98 82 66 30 12");
    }
    algoritmo_35{
    print("11 99 83 67 31 13");
    }

36. Precondición: true
    Datos de salida: un string
    Salida estándar: la palabra "Hola Mundo!"
    Postcondición: imprime "Hola Mundo!"

37. Datos de entrada: un positivo: n
    Precondición: true
    Datos de salida: un natural: total
    Postcondición: se obtiene la suma de todos los números del 1 al n

38. El enunciado de este programa es ambiguo, estas son sus dos interpretaciones:
    Datos de entrada: dos enteros: a, b
    Precondición: b /= 0
    Datos de salida: dos enteros: q, r
    Postcondición: a = b * q + r ^ r < b
    o
    Datos de entrada: dos enteros: b, a
    Precondición: a /= 0
    Datos de salida: dos enteros: q, r
    Postcondición: b = a* q + r ^ r < a

39. Datos de entrada: una secuencia de caracteres
    Entrada estándar: dicha secuencia
    Precondición: true
    Datos de salida: cinco enteros: a, e, i, o, u
    Salida estándar: dichos enteros
    Postcondición: imprime el nº de apariciones de a, e, i, o, u, en la secuencia, respectivamente

40. Datos de entrada: un real: t en horas
    Precondición: t > 0
    Datos de salida: tres enteros: h, min, sg
    Postcondición: imprime la hora t en horas h, minutos min y segundos sg

41. Datos de entrada: un entero: n
    Precondición: n > 0
    Datos de salida: un booleano, p
    Postcondición: devuelve true si y solo si n es primo, es decir, sus únicos divisores son 1 y n

42. Datos de entrada: un entero: n
    Precondición: n > 0
    Datos de salida: un booleano, p
    Postcondición: devuelve true si y solo si n es pseudoprimo, es decir,
    tiene otros dos divisores distintos de 1 y N y estos divisores son primos

43. El enunciado de este programa es ambiguo, estas son sus dos interpretaciones:
    Datos de entrada: tres reales: a,b,c
    Precondición: 0 <= a,b,c < 360
    Datos de salida: un booleano, t y un string, s
    Postcondición: devuelve true si la suma de los ángulos a + b + c = 180, y si t es true, entonces s adquiere uno de estos valores:
    - a, b, c < 90 -> s = “Acutángulo”
    - Uno de los ángulos mide 90 -> s = “Rectángulo”
    - Uno de los ángulos mide más de 90 -> s = “Obtusángulo”

    o
    Datos de entrada: tres reales: a,b,c
    Precondición: 0 <= a,b,c < 2π
    Datos de salida: un booleano, t y un string, s
    Postcondición: devuelve true si la suma de los ángulos a + b + c = π, y si t es true, entonces s adquiere uno de estos valores:
    - a, b, c < π/2 -> s = “Acutángulo”
    - Uno de los ángulos mide π/2 -> s = “Rectángulo”
    - Uno de los ángulos mide más de π/2 -> s = “Obtusángulo”

44. Datos de entrada: tres reales: a,b,c
    Precondición: a,b,c > 0
    Datos de salida: un booleano, t y un string, s
    Postcondición: devuelve true si la suma de los dos lados menores es mayor que la longitud del lado mayor, y si t es true, entonces s adquiere uno de estos valores:
    - Todos los lados iguales -> s = “Equilátero”
    - Dos lados de tres son iguales -> s = “Isósceles”
    - Todos sus lados son desiguales -> s = “Escaleno”

45. El enunciado de este programa es ambiguo, estas son sus dos interpretaciones:
    Datos de entrada: un natural, 20 patos
    Precondición: true
    Datos de salida: dos enteros, patas y picos
    Postcondición: si cada pato tiene un pico y dos patas, entonces se devolverán 20 picos y 40 patas
    o
    Datos de entrada: un natural, 2 patos
    Precondición: true
    Datos de salida: dos enteros, patas y picos
    Postcondición: si cada pato tiene un pico y dos patas, entonces se devolverán 2 picos y 4 patas

46. Datos de entrada:tres reales: a, b, c
    Entrada estándar: dichos reales
    Precondición: true
    Datos de salida: un real: mayor
    Salida estándar: dicho real
    Postcondición: mayor adquiere el valor mayor entre a, b, c
47. Datos de entrada: un entero: n
    Precondición: n > 0
    Datos de salida: un entero, fact
    Postcondición: fact recibe el valor del factorial de n (n! = n * (n-1) * … * 2 * 1)

48. Precondición: true
    Datos de salida: una secuencia de asteriscos
    Salida estándar: dicha secuencia
    Postcondición: una cadena de asteriscos de longitud igual al número de primos entre 1 y 100

49. Datos de entrada: un entero n
    Precondición: true
    Datos de salida: un entero cnt
    Postcondición: cnt recibe el número de cifras que contiene n

50. Datos de entrada: una secuencia de caracteres
    Entrada estándar: dicha secuencia
    Precondición: todos los caracteres de la secuencia son letras minúsculas, salvo el último que es un espacio en blanco
    Datos de salida: la secuencia transformada
    Salida estándar: dicha secuencia
    Postcondición: se devuelve dicha secuencia transformada , de forma que cada carácter se sustituye por el que esté situado tres posiciones a la derecha. El principio del abecedario se enrosca con el final de forma que dicho recorrido sea circular. El carácter espacio, el último, no se transforma
51. (*)

Entrada	Descripción del caso	Salida
(5, 2)	Pares de igual signo, ambos números positivos	(2, 1)
(-5, -2)	Pares de igual signo, ambos números negativos	(1, -3)
(5, -2)	Pares de distinto signo, dividendo positivo, divisor negativo	(-4, -3)
(-5, 2)	Pares de distinto signo, dividendo negativo, divisor positivo	(-2, -1)
(0, 2)	Dividendo nulo, divisor positivo	(0, 0)

52. (*)

Entrada	Descripción del caso	Salida
0.5	Radio dado entre 0 y 1	0.25π
1	Radio dado igual a 1	π
2	Radio dado mayor que uno	4π

53. (*)

Entrada	Descripción del caso	Salida
(1, 0, -1)	Delta (Δ) es mayor que 0	(1, -1)
(1, 2 , 1)	Delta (Δ) es igual a 0	(-1, -1)

54. (**)

Entrada	Descripción del caso	Salida
1	Posición del día Lunes	L
2	Posición del día Martes	M
3	Posición del día Miércoles	X
4	Posición del día Jueves	J
5	Posición del día Viernes	V
6	Posición del día Sábado	S
7	Posición del día Domingo	D

55. (**)

Entrada	Descripción del caso	Salida
(25, 12)	Días de diciembre a partir del 21	Invierno
(14, 2)	Días de enero y febrero	Invierno
(3, 3)	Días de marzo antes del 21	Invierno
(23, 3)	Días de marzo a partir del 21	Primavera
(6, 4)	Días de abril y mayo	Primavera
(2, 6)	Días de junio antes del 21	Primavera
(24, 6)	Días de junio a partir del 21	Verano
(7, 7)	Días de julio y agosto	Verano
(11, 9)	Días de septiembre antes del 21	Verano
(30, 9)	Días de septiembre a partir del 21	Otoño
(31, 10)	Días de octubre y noviembre	Otoño
(6, 12)	Días de diciembre antes del 21	Otoño

56. (**)

Entrada	Descripción del caso	Salida
4.2	Nota entre 0 y 5	Suspenso
6.1	Nota entre 5 y 7	Aprobado
7.5	Nota entre 7 y 9	Notable
9.3	Nota entre 9 y 10	Sobresaliente

57. (**)

Entrada	Descripción del caso	Salida
1	Hora exacta	(1, 0, 0)
1.5	Hora no exacta, pero el número sin la coma es múltiplo de 60	(1, 30, 0)
1.21	Resto de horas no exactas	(1, 12, 36)

58. (**)

Entrada	Descripción del caso	Salida
6	El supremo es un número positivo menor que el m.c.m(3, 5) = 15	1
45	El supremo es mayor o igual a 15	1, 15, 30, 45

59. (***)

Entrada	Descripción del caso	Salida
1	Unidad	False
17	Número primo	False
21	Número compuesto de dos factores primos	True
12	Número compuesto de más de dos factores primos	False

60. (***)

Entrada	Descripción del caso	Salida
-1	Secuencia vacía	0
5, -1	Secuencia de un número	5
3, 4, -2, 1, -1	Secuencia de más de un número	6

61. (***)

Entrada	Descripción del caso	Salida
(1, 1, 3)	La suma de los dos lados menores es menor o igual que la longitud del lado mayor	FALSE
------	La suma de los dos lados menores es mayor que la longitud del lado mayor	TRUE
(3, 3, 3)	Tres lados iguales	Equilátero
(3, 3, 4)	Dos lados iguales entre sí y el tercero desigual	Isósceles
(6, 4, 3)	Tres lados desiguales	Escaleno

62. (****)

Entrada	Descripción del caso	Salida
135	El nº no tiene dígitos pares	(0, 0)
2345	El número tiene al menos un dígito par pero no aparecen más de dos dígitos pares seguidos	(2, 1)
224568	El número tiene al menos un dígito par y hay una secuencia de al menos dos dígitos pares seguidos	(5, 3)

63. :

- (*) not a = false
- (*) a or b = true
- (*) a and b = false
- (*) a and not b = true
- (*) a + b = ERROR (Sumar booleanos)
- (*) f + g = 7
- (*) c – d = 8
- (*) d * f = -10
- (*) f / g = 2
- (*) f / e = ERROR (División por cero)
- (*) c % f = 1
- (*) i % -d = ERROR (Mod con Float)
- (*) g + i = ERROR (Suma int con float)
- (*) 3.0 * i = 3.0
- (*) h + 2 = ERROR (Suma int con float)
- (**) ord(j) = 121
- (**) ord('j') = 106
- (**) isAlpha(e) = ERROR (isAlpha con int)

- (**) isAlpha('e') = true
- (**) isAlpha(l) = false
- (**) upper(l) = ERROR ('2' not isAlpha)
- (**) upper(j) = 'Y'
- (**) pred(k) = 'd'
- (**) isDigit(k) = false
- (**) succ(m) = 'A'
- (**) c * d + f / g = -10
- (**) 2.5 * (2 .0 + i) = 7.5
- (**) a > b = ERROR (Comparar booleanos)
- (**) e < f = true
- (**) g + c >= 5 = true
- (**) (i + h)/2.0 > 0.5 = false
- (**) 'y' <> j = false
- (**) m > k = false
- (**) k < j = true
- (**) 'k' < 'j' = false
- (***) (2*c % f = g) xor isDigit(l) = false
- (***) succ (chr(ord('j') + g)) = 'm'
- (***) upper(pred(k)) = 'D'
- (***) isAlpha(j) and (k < succ('k')) = true
- (***) 3.5*i + 2.0 > c – d (***) = ERROR (comparar int con float)
- (***) not isDigit(chr(ord('+') + f)) = false
- (****) c + d / g – (ord(k) – ord (m)) / c + ord(succ(chr(sqrt(10*g + f) + f*c))) = 35
- (****) (-f^g > -ord(m)) and isAlpha(chr(10^-d + ord(l) + c*d – ord(pred('2')))) = true

64. :

- (*) pred(Hoy) = M
- (*) pred(Ayer) = ERROR (pred(first(Días))
- (*) succ(Hoy) = J
- (*) succ(Ayer) = M
- (*) isAlpha(Letra) = true
- (*) lower(Letra) = 'e'
- (*) n – 1 = ERROR (valor fuera de rango)
- (**) Letra > 'D' = true
- (**) Letra < J = ERROR (comparar Alfabeto con Días)
- (**) L < J = true
- (**) 'L' < 'J' = false
- (**) n in int = true
- (**) 'T' in Alfabeto = true
- (**) 't' in Alfabeto = false
- (**) 'X' in Días = false
- (**) X in Días = true
- (***) pos('S') = 19
- (***) pos(S) = 6

- (***) n + ord(Letra) = 70
- (***) val(ord(Letra) / 12) para Días = V
- (***) val(ord(Letra) / 12) para Alfabeto = 'E'
- (***) first(Alfabeto) = Letra = false
- (***) last(Alfabeto) <> 'z'= true
- (****) pred(val(pos(Hoy) - 1)) para Alfabeto = 'A'

Desde este punto de partida hasta nueva instrucción, en los ejercicios de programación, la solución a estos está parcialmente completa por motivos de simplificación. Hay que habilitar aserciones, inicializar variables de entrada y escribir una aserción en donde se verifica que el valor de los datos de salida es el esperado con los datos de entrada dados. Se muestra la solución completa como ejemplo en el ejercicio 65.

65. (*)
```
PROGRAM EcuacionRecta;
{$ASSERTIONS ON}
VAR
        y, m, x, n: REAL;
BEGIN
        m := 2.0;
        x := 3.0;
        n := 1.0;
        y := m*x + n;
        assert(y = 7.0);
END.
```
66. (*)
```
PROGRAM MediaDiezNotas;
VAR
        N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, suma, media: REAL;
BEGIN
        suma := N1 + N2 + N3 + N4 + N5 + N6 + N7 + N8 + N9 + N10;
        media := suma / 10.0;
END.
```
67. (*)
```
PROGRAM MediaPonderadaTresNotas;
VAR
        N1, N2, N3, media: REAL;
BEGIN
        media := 0.2 * N1 + 0.3 * N2 + 0.5 * N3;
END.
```
68. (*)
```
PROGRAM SumaPar;
VAR
        A, B: INTEGER;
        esPar: BOOLEAN;
BEGIN
        esPar := (A + B) mod 2 = 0;
END.
```

69. (*)
```
PROGRAM SumaParSinSumar;
VAR
        A, B: INTEGER;
        esPar: BOOLEAN;
BEGIN
        esPar := (A mod 2 = 0 and B mod 2 = 0) or (A mod 2 = 1 and B mod 2 = 1);
END.
```
70. (*)
```
PROGRAM EsMultiploSatanico;
VAR
        N: INTEGER;
        esMulSatanico: BOOLEAN;
BEGIN
        esMulSatanico:= (N mod 6 = 0 and N mod 66 = 0) or N mod 666 = 0;
END.
```
71. (*)
```
PROGRAM TrianguloAritmetico;
VAR
        A, B, C, PAR1, PAR2, RESULT: INTEGER;
BEGIN
        PAR1 := A + B;
        PAR2 := B + C;
        RESULT := PAR1 + PAR2;
END.
```
72. (**)
```
PROGRAM VolumenCilindro;
CONST
        PI : REAL := 3.1416;
VAR
        Abase, r, h, V: REAL;
BEGIN
        Abase := PI*r*r;
        V := Abase * h;
 END.
```

73. (**)
```
PROGRAM AreaTanqueCilindrico;
CONST
        PI : REAL := 3.1416;
VAR
        Abase, Acilindro, Atotal, r, h, V: REAL;
BEGIN
        Abase := PI*r*r;
        Acilindro := 2*PI*r*h;
        Atotal := Abase + Acilindro;
END.
```

74. (**)
```
PROGRAM Porcentajes;
VAR
        A, B, C, suma, porcA, porcB, porcC: FLOAT;
BEGIN
        suma := A + B + C;
        porcA := A * 100.0 / suma;
        porcB := B * 100.0 / suma;
        porcC := C * 100.0 / suma;
END.
```
75. (**)
```
PROGRAM Intercambiar;
VAR
        A, B, aux: INTEGER;
BEGIN
        aux := A;
        A := B;
        B := aux;
END.
```
76. (**)
```
PROGRAM DescomperCuatroCifras;
VAR
        N, M, C, D, U: INTEGER;
BEGIN
        M := N div 1000;
        N := N mod 1000;
        C := N div 100;
        N := N mod 100;
        D := N div 10;
        N := N mod 10;
        U := N;
END.
```
77. (**)
```
PROGRAM EsDigito;
VAR
        Ch: CHARACTER;
        esDigito: BOOLEAN;
BEGIN
        esDigito := Ch >= '0' and Ch <= '9';
END.
```
78. (**)
```
PROGRAM EsAlfabetico;
VAR
        Ch: CHARACTER;
        esAlfabetico: BOOLEAN;
BEGIN
        esAlfabetico := (Ch >= 'a' and Ch <= 'z') or (Ch >= 'A' and Ch <= 'Z');
END.
```

79. (**)

```
PROGRAM SwapCaracteres;
VAR
        X, Y, aux: CHARACTER;
BEGIN
        aux := X;
        X := Y;
        Y := aux;
        X := chr(ord(X) - 32);
        Y := chr(ord(X) + 32);
END.
```

80. (***)

```
PROGRAM SumaDigitos;
TYPE
        Digito = '0'..'9';
VAR
        A, B, suma: Digito;
        intA, intB, intSuma: INTEGER;
        desborde : BOOLEAN;
BEGIN
        intA := ord(A) – 48;
        intB := ord(B) – 48;
        intSuma := (intA + intB) mod 10;
        suma := chr(intSuma + 48);
        desborde := (intA + intB) >= 10;
END.
```

81. (***)

```
PROGRAM EsFonemaTrabado;
VAR
        x, y: CHARACTER;
        esFonTrab: BOOLEAN;
BEGIN
        esFonTrab := (x = 'b' or x = 'c' or x = 'd' or x = 'f' or x = 'g' or x = 'p' or x = 't') and (y = 'l'
        and not (x = 'd' or x = 't') or y = 'r');
END.
```

82. (***)

```
PROGRAM ResolverEcuacionCuadratica;
VAR
        a, b, c, delta, raiz_delta, x_1, x_2: REAL;
BEGIN
        delta := b*b – 4*a*c;
        raiz_delta := 1.0 + 0.5 * (delta – 1) – 0.125 * (delta – 1) * (delta – 1);
        x_1 := (-b + raiz_delta) / 2*a;
        x_2 := (-b – raiz_delta) / 2*a;
END.
```

83. (***)

```
PROGRAM DeBinomicaAPolar;
VAR
        a, b, r, r_2, theta, arg: REAL;
BEGIN
        r_2 := a*a + b*b;
        r := 1.0 + 0.5 * (r_2 – 1) – 0.125 * (r_2 – 1) * (r_2 – 1);
        theta := b/a;
        arg := (theta) + (theta)* (theta)* (theta) / 3;
END.
```

84. (***)

```
PROGRAM TiempoIncorporacion;
VAR
        v, vf, x, a, t: REAL;
BEGIN
        a := (vf*vf – v*v) / 2*x;
        t := (vf – v) / a;
END.
```

85. (***)

```
PROGRAM LongitudApartadero;
VAR
        v, vo, t, a, x: REAL;
BEGIN
        vo := 1.3 * v;
        a := - vo / t;
        x := vo*t + 0.5*a*t*t;
END.
```

86. (****)

```
PROGRAM TiempoTranscurrido;
VAR
        h1, m1, s1, h2, m2, s2, t1, t2, dt, hr, mr, sr: INTEGER;
BEGIN
        t1 := h1*3600 + m1*60 + s1;
        t2 := h2*3600 + m2*60 + s2;
        dt := t2 – t1;
        sr := dt mod 60;
        dt := dt div 60;
        mr := dt mod 60;
        hr := dt div 60;
END.
```

87. (****)

```
PROGRAM TiempoTranscurridoEnSexagesimal;
VAR
        h1, m1, s1, h2, m2, s2, hr, mr, sr, llevadaS, llevadaM: INTEGER;
BEGIN
        llevadaS := (s2 < s1) * 1;
        sr := (s2 + 60 * llevadaS) – s1;
        m2 := m2 – llevadaS;
        llevadaM := (m2 < m1) * 1;
        mr := (m2+60*llevadaM)–m1;
        h2 := h2 – llevadaM;
        hr := h2 – h1;
END.
```

88. (****)

```
PROGRAM Decoder7Segmentos;
VAR
        n: INTEGER;
        a, b, c, d, e, f, g: BOOLEAN;
BEGIN
        a := not (n = 1 or n = 4);
        b := (n >= 2 and n <= 9) and not (n = 7);
        c := not (n = 1 or n = 4 or n=7);
        d := n = 0 or ((n >= 4 and n<=9) and not (n = 7));
        e := n=0 or n=2 or n=6 or n=8;
        f := not (n = 5 or n = 6);
        g := not n = 2;
END.
```

89. (****)

```
PROGRAM PuentesHashiValidos;
VAR
        x1, y1, x2, y2, x3, y3, x4, y4, hor1, hor2, xv, yh, xizq, xdcha, yup, ydown: INTEGER;
        ptesValidos: BOOLEAN;
BEGIN
        hor1 := (y1 = y2) * 1;
        hor2 := (y3 = y4) * 1;
        xv := x3 * hor1 + x1 * hor2;
        yh := y1 * hor1 + y3 * hor2;
        xizq := x1 * hor1 + x3 * hor2;
        xdcha := x2 * hor1 + x4 * hor2;
        yup := y4 * hor1 + y2 * hor2;
        ydown := y3*hor1 + y1*hor2;
        ptesValidos := (hor1 = hor2) or (xizq < xv and xv < xdcha and ydown < yh and yh <
        yup);
END.
```

90. (*****)

```
PROGRAM HoraEpoch;
VAR
        h, min, sg, d, mes, anno,
        esBisiesto, bisiestos, diasMes, días, horaepoch: INTEGER;
BEGIN
        diasMes := 31*(mes=2) + (31+28)*(mes=3) + (31+28+31)*(mes=4) +
        (31+28+31+30)*(mes=5) + (31+28+31+30+31)*(mes=6) +
        (31+28+31+30+31+30)*(mes=7) + (31+28+31+30+31+30+31)*(mes=8) +
        (31+28+31+30+31+30+31+31)*(mes=9) +
        (31+28+31+30+31+30+31+31+30)*(mes=10) +
        (31+28+31+30+31+30+31+31+30+31)*(mes=11) +
        (31+28+31+30+31+30+31+31+30+31+30)*(mes=12);
        esBisiesto := ((anno mod 4 = 0 and not anno mod 100 = 0) or anno mod 400 = 0) * 1;
        días := diasMes + d + esBisiesto * (mes > 2);
        bisiestos := (anno – 1970) div 4 – (anno – 1970) div 10 + (anno – 1970) div 400;
        días := (anno – 1970) * 365 + bisiestos + (dias – 1);
        horaepoch := dias * 24 * 3600 + h * 3600 + min * 60 + sg;
END.
```

BIBLIOGRAFÍA

Aguilar, Luis Joyanes. *Fundamentos de la programación. Algoritmos y Estructuras de datos.* s.l. : McGraw Hill.

Alonso, Fernando Antón. *Cómo programar en lenguaje C.* s.l. : Prensa Técnica.

Alzate, Alonso Tamayo. *Programación Estructurada. Un enfoque algorítmico.* s.l. : Universidad Nacional de Colombia.

Andrés Marzal, Isabel Gracia. *Introducción a la Programación con Python.* s.l. : Universitat Jaume I.

Arantza Díaz de Illaraza, Kepa Sarasola. *Oinarrizko Programazioa. Ariketa-bilduma.* s.l. : Udako Euskal Unibertsitatea.

Barnes, J.G.P. *Programación en ADA.* s.l. : DÍAZ DE SANTOS S.A.

Bowen, Kenneth A. *Speaking Pascal: A Computer Language Primer.* s.l. : Hayden.

Fernández, Tomás Antonio Pérez. *Programación Básica. Manual de laboratorios.*

Forsythe, Keenan, Organick, Stenberg. *Lenguajes de Diagramas de Flujo.* s.l. : Limusa.

ÍNDICE ALFABÉTICO