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.

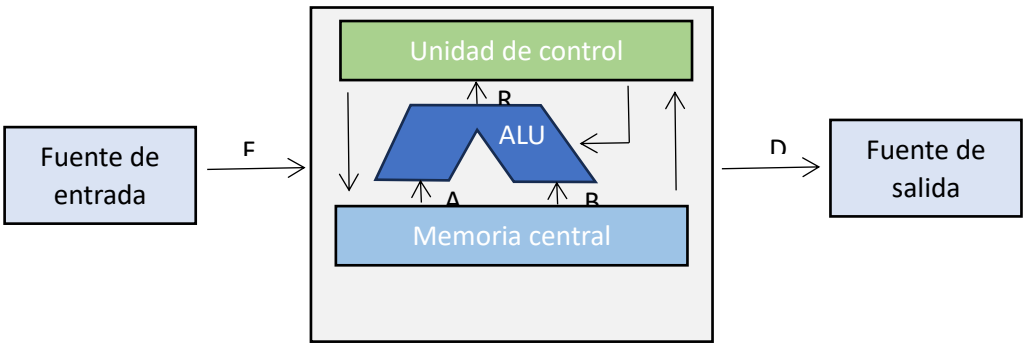
Para terminar este tema, me gustaría explicar un poco más detalladamente cómo el ordenador es capaz de ejecutar un programa a través de los distintos componentes del ordenador (hardware). Para ello, asumimos que nuestro procesador sigue la **arquitectura de Von Neumann**.

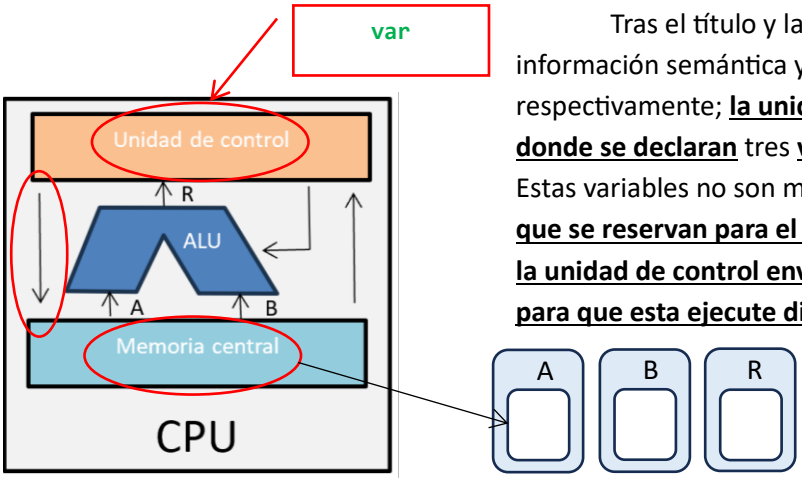
Este diseño es bastante más simplificado de lo que sería la arquitectura del computador en realidad, pero lo suficiente como para comprender el mecanismo de la programación. Pondremos como ejemplo otro programa en lenguaje Pascal.

```
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.

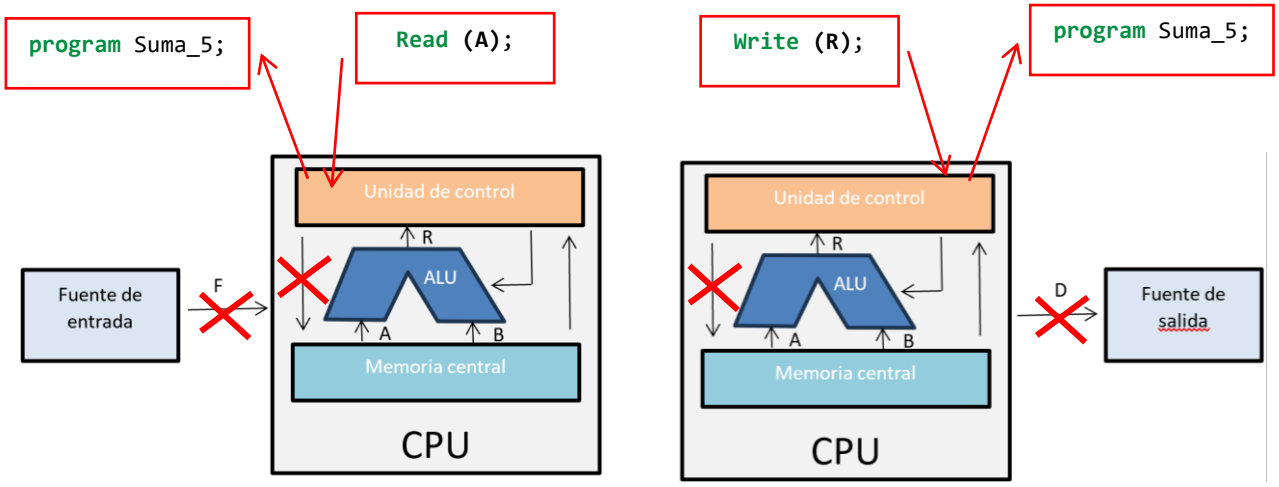




Tras el título y la especificación, que es más bien información semántica y descriptiva para el desarrollador, respectivamente; **la unidad de control recibe la instrucción donde se declaran** tres **variables**, A, B y R de tipo entero. Estas variables no son más que **huecos de memoria vacíos que se 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**.

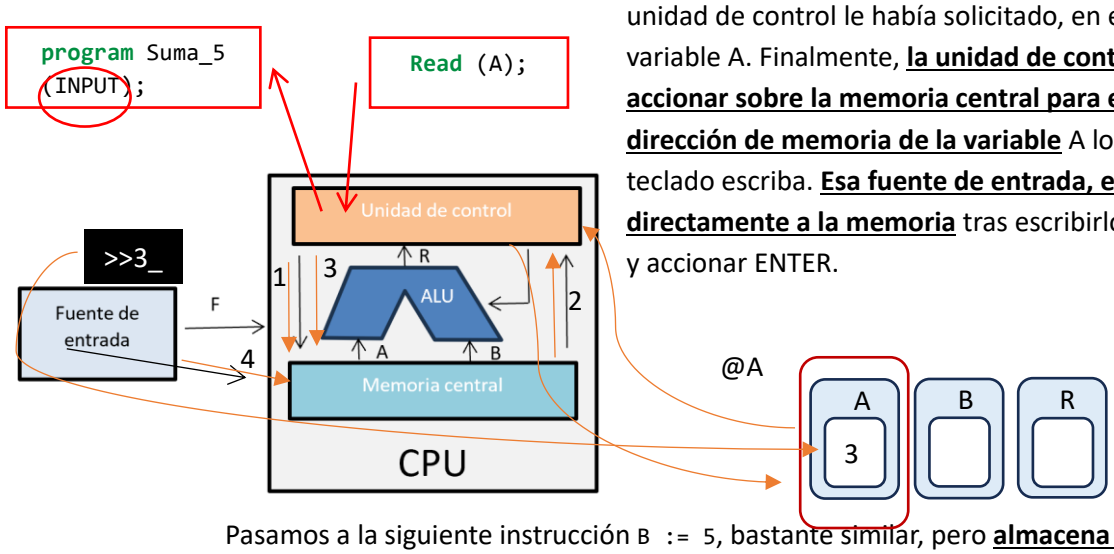
A continuación, se procede a ejecutar el algoritmo del programa. En este caso, debido a la sintaxis del lenguaje, la unidad de control lo procesa porque recibe la palabra reservada `begin`, la cual indica el inicio del algoritmo.

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**.



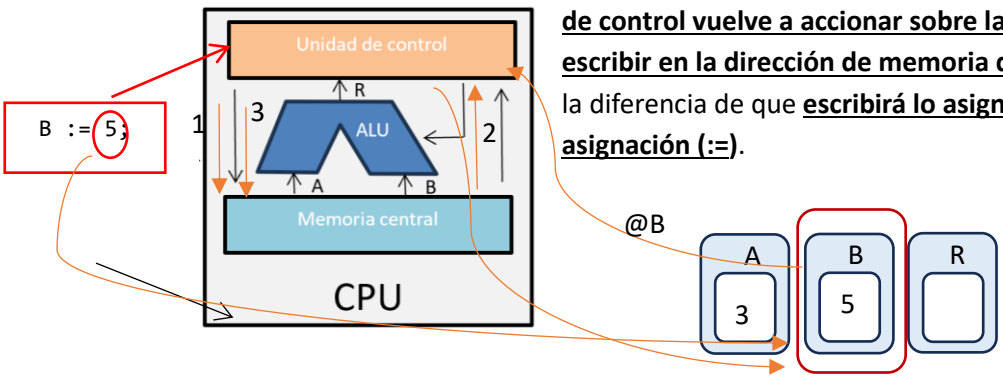
Lo segundo, **al introducir un dato por el teclado, hay que asegurarse de que lo que se introduzca pertenece al tipo de dato que se solicita**. Así mismo, si se pide un entero no se pueden introducir ni números decimales, ni letras, ni oraciones ni nada distinto a un número entero. Si no se hiciera,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



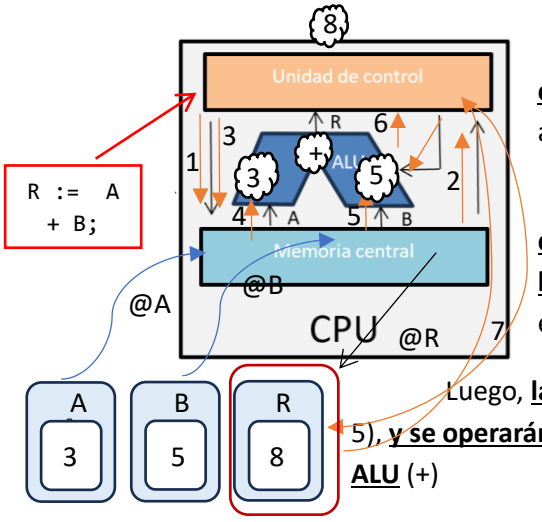
Pasamos a la siguiente instrucción `B := 5`, bastante similar, pero **almacena el dato** directamente dada una instrucción que asigna el valor 5 a la variable B, **por acción del código en sí** y no por fuentes externas al procesador.

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.

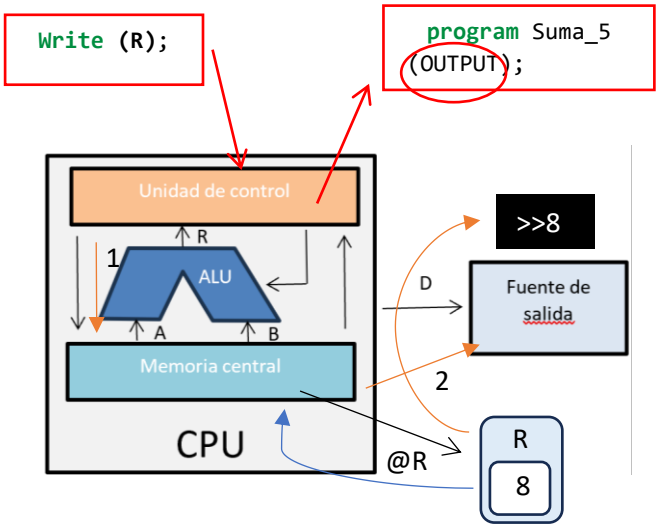
La memoria envía a la unidad de control la dirección de memoria de la variable de destino (R) **y busca las direcciones de memoria de las variables fuente** (A y B), es decir, los operandos.

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.

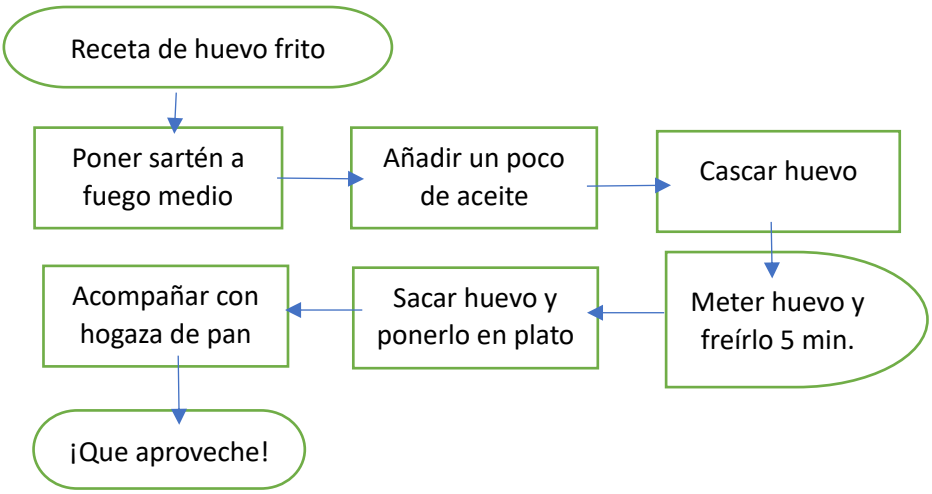
La importancia de la abstracción radica en **evitar la ambigüedad que se pueda cometer usando el lenguaje natural**, por ello, no es buena idea diseñar algoritmos con nuestras propias palabras. Por ejemplo, la frase "he visto a una persona con unos prismáticos" puede tener dos interpretaciones distintas.

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:

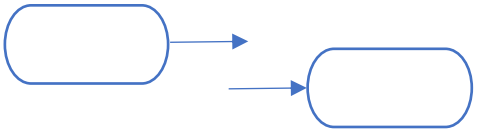
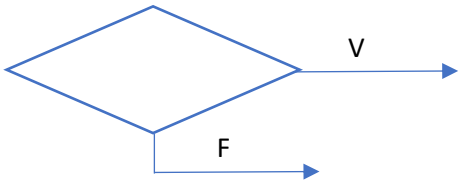


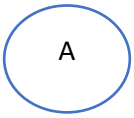
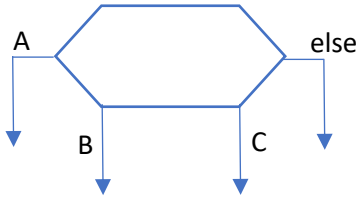
Ambas formas de expresión de la receta simbolizan el mismo algoritmo. Además, este tipo de gráficos no sólo se utilizan en el contexto de **la programación**, sino que también tiene otras utilidades en otras disciplinas como **la economía, la tecnología industrial y la psicología**.

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

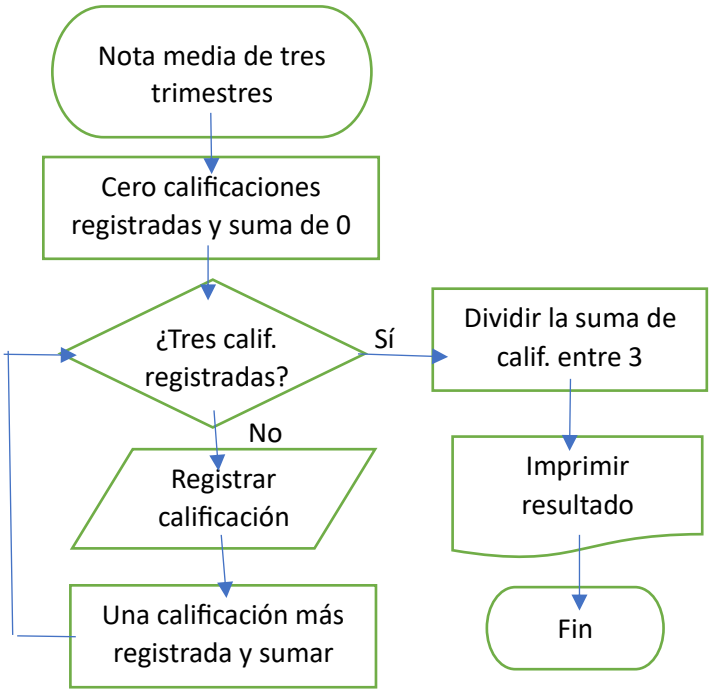
	bloque sobre el que se quiere comentar, y el bloque se coloca aparte de la secuencia.
	***Bloque de proceso predefinido***. Se usará a partir del Tema 9 para indicar Subprogramas. Se explicará en su respectivo tema.
	***Conectores***. Cuando las líneas de flujo son largas o confusas, se utilizan dos 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.

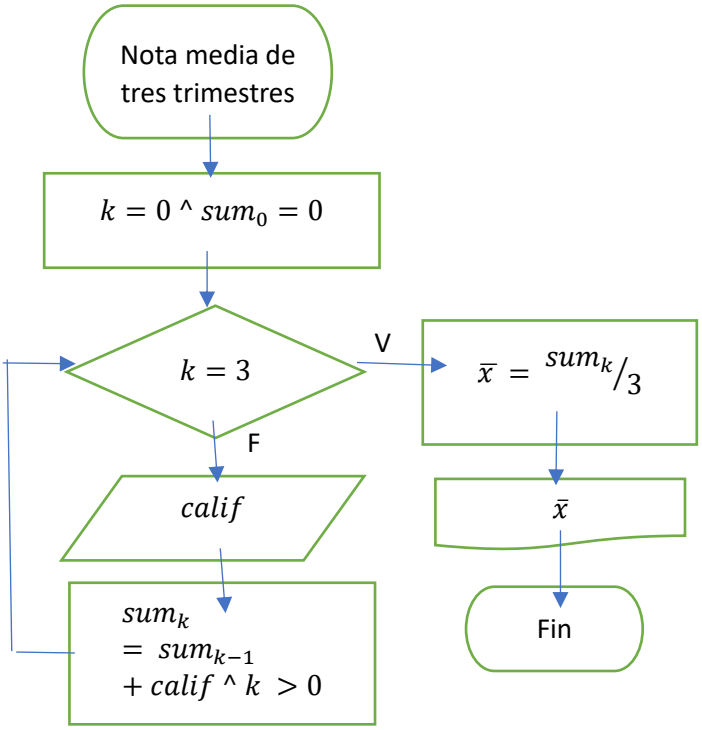
Es más, este diagrama todavía puede mejorar si recurrimos a un **lenguaje abstracto** como el **algebraico** o el **lenguaje formal de diagramas de flujo**. El lenguaje algebraico ya lo conocemos, es el que permite que en un problema como por ejemplo "La suma de edades de los tres hijos es 30. Las edades 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**

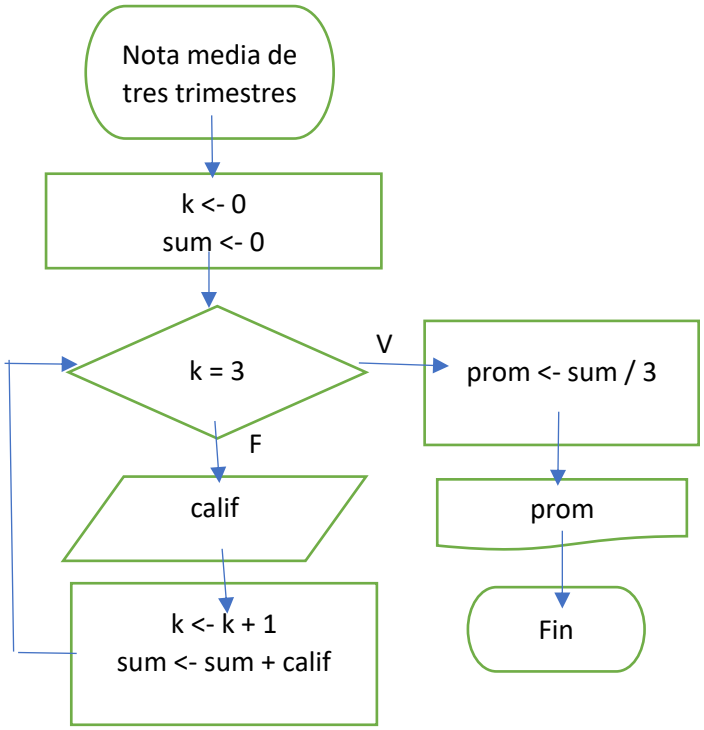
Este lenguaje **destaca por su parecido a los primeros lenguajes de programación creados, por su simplicidad y su linealidad**. Sus reglas son las siguientes:

- 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:

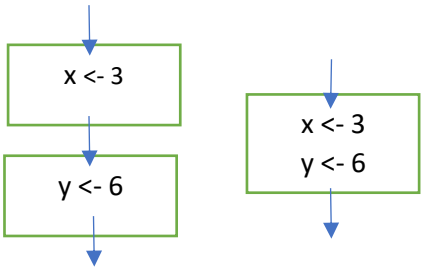


Este lenguaje formal nos expone la principal **diferencia entre la igualdad y la asignación** en el contexto de la programación, y que su concepto no es el matemático.

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.

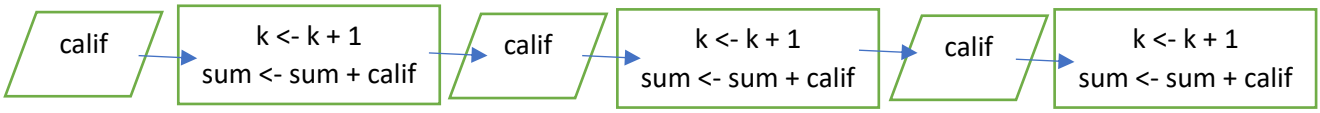
Esto permite expresiones como k <- k + 1, es decir, "asigno a la variable k el valor que tiene actualmente k más uno" Hay que tener en cuenta también que una expresión algebraica como x = x + 1 no tiene solución, por lo que por reducción al absurdo no tiene sentido hablar de igualdad en las variables,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

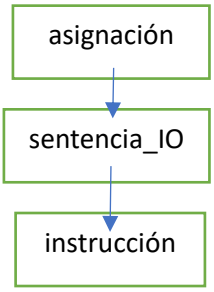
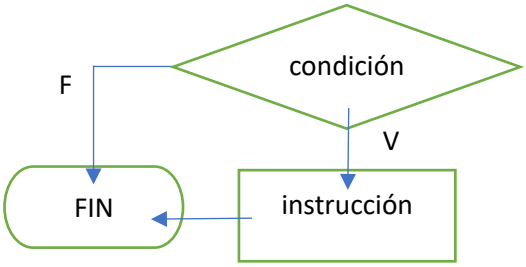
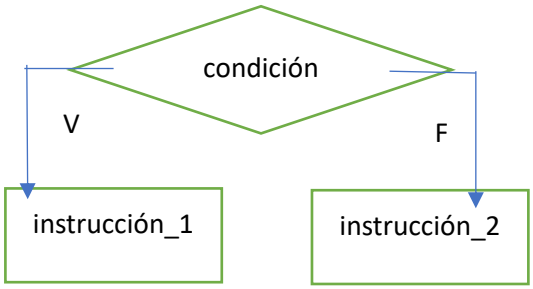
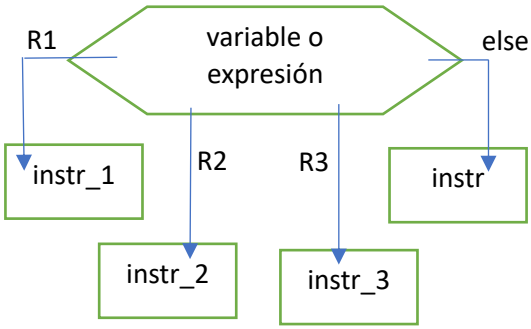
En segundo lugar, **el bucle**, que destaca por su flecha estilo [corchete, **sustituye a una sucesión de repetidas asignaciones**, ahorrando bastante espacio. En el ejemplo anterior, el bucle podría sustituirse ineficientemente por:

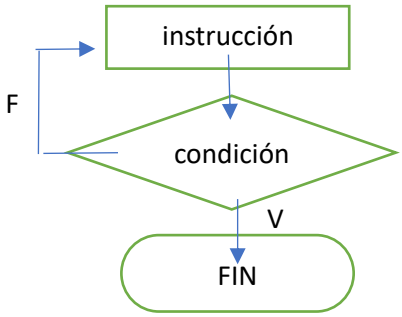
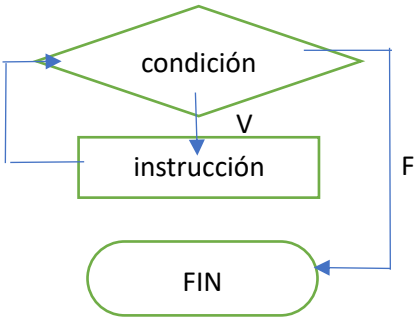
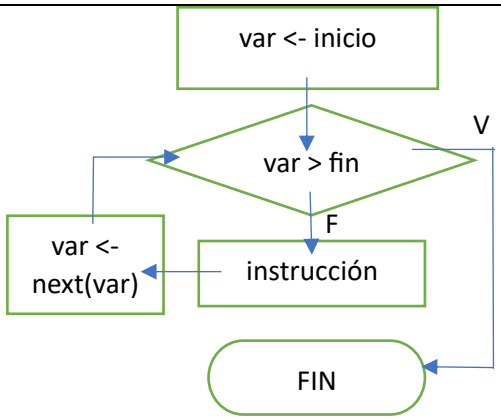


1.2 Pseudocódigo

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><pre>
<asignación>::= <variable> := <valor> |
<expresión> ;
<sentencia_IO>::= scan() | print() ;
<instrucción>::= <asignación> |
<sentencia_IO>;
</pre></td>
<td><pre>
<condición>::= <expresión_booleana>;
<bloque_if>::=
if (<condición>) then
    <instrucción>; {Se cumple la cond.}
</pre></td>
</tr>
<tr>
<td>Estructura secuencial.
Es aquella que ejecuta una sucesión de instrucciones secuencialmente, ejecutándolas todas y sólo una vez.</td>
<td>Estructura selectiva simple.
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>
<tr>
<td><pre>
<bloque_if_else>::=
if (<condición>) then
    <instrucción_1>; {Se cumple la cond}
else
    <instrucción_2>; {No se cumple}
</pre></td>
<td><pre>
<bloque_cases>::=
cases of <variable> | <expresión>
     <rango_1>: <instrucción_1>;
     <rango_2>: <instrucción_2>;
     <rango_3>: <instrucción_3>;
     else: <instrucción>; {otros rangos}
</pre></td>
</tr>
</table>

<table>
<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><pre><bloque_repeat>::=
repeat
    <instrucción>;
until (<condición>);</pre></td>
<td><pre><bloque_while>::=
while (<condición>) do
    <instrucción>;</pre></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><pre><bloque_for>::=
for <var> in range <inicio> to <fin> do
    <instrucción>;</pre></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:

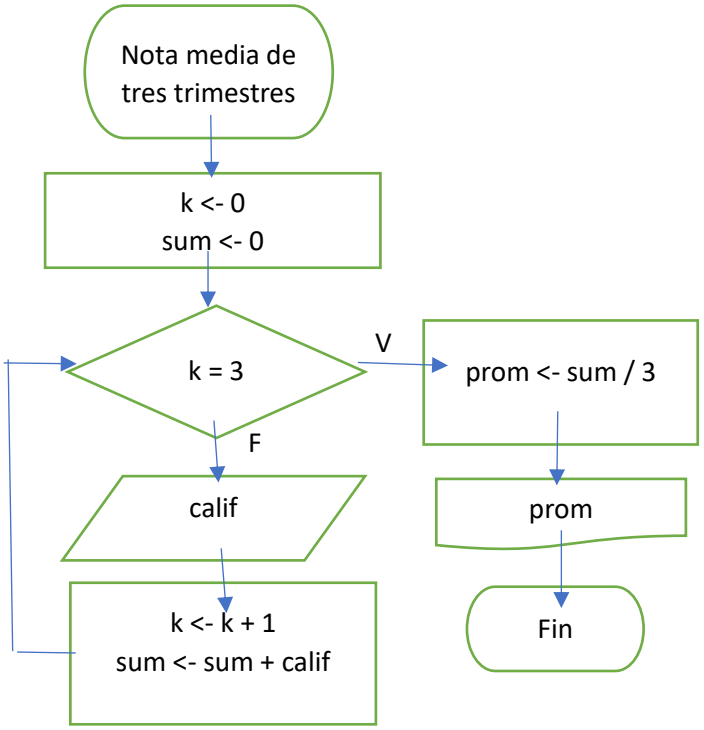
```
i:=2;
while (i<=size) do
    if (v[i-1]>v[i]) then
        aux:=v[i-1];
    end if;
end while;
v[i-1]:=v[i];
v[i]:=aux;
i:=i+1;
```

```
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.

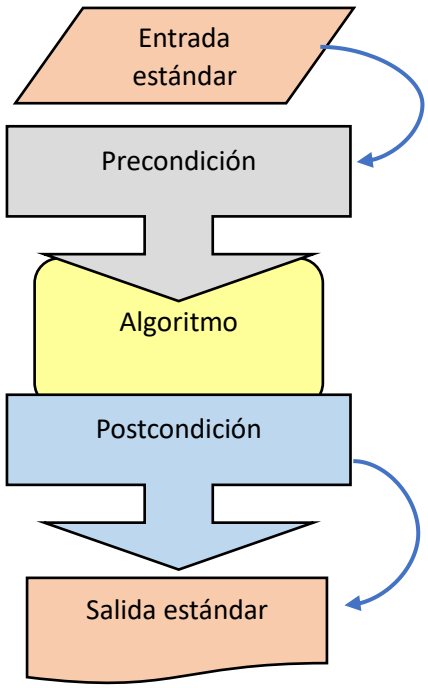
Cuando se diseña un producto **hay que tener en cuenta todos los casos de uso que se pueden presentar**; por ejemplo, alguien podría decidir conectar el aparato a una red de 300V cuando se indica explícitamente en el manual que no se puede conectar a corrientes superiores a 250V. **Las especificaciones** (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:

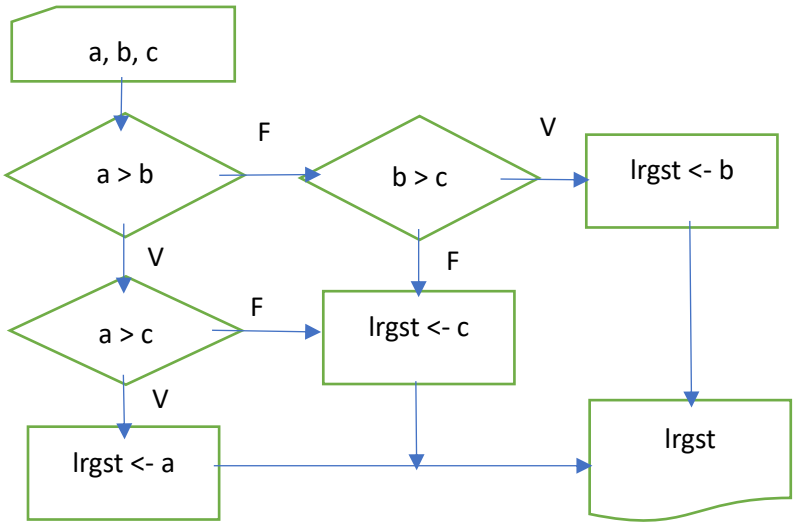
29. (*) El programa calcula la raíz cuadrada de x menos y
30. (*) El programa calcula el área de un triángulo equilátero dado el lado
31. (*) El programa calcula la diferencia de dos números
32. (**) El programa devuelve una letra correspondiente al día de la semana en castellano dado un número del 1 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
```

48. (****)

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

```
    fact := fact * x;
    x := x + 1;
```

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)