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.

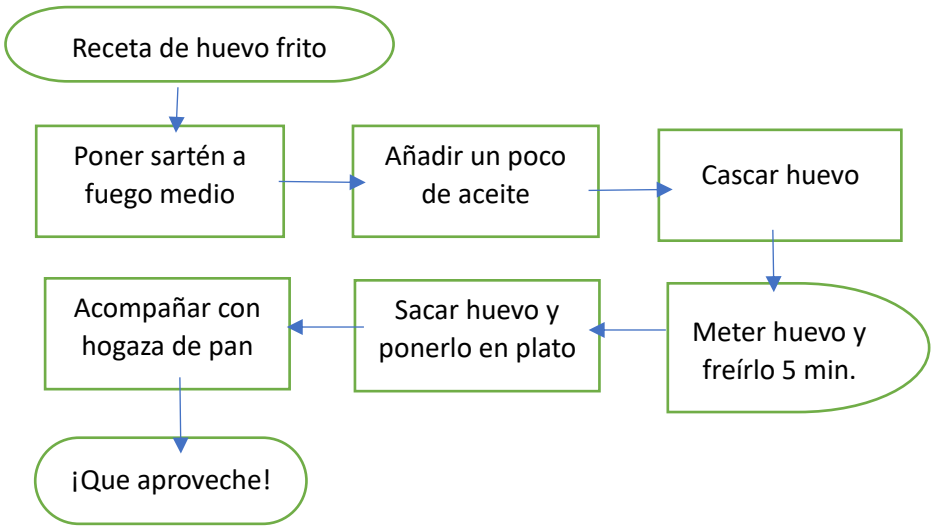
La importancia de la abstracción radica en **evitar la ambigüedad que se pueda cometer usando el lenguaje natural**, por ello, no es buena idea diseñar algoritmos con nuestras propias palabras. Por ejemplo, la frase "he visto a una persona con unos prismáticos" puede tener dos interpretaciones distintas.

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:

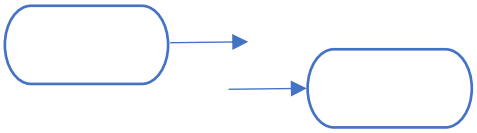
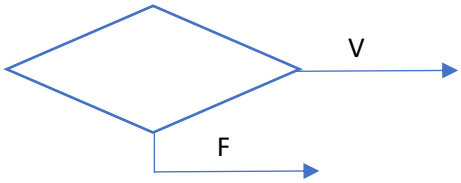


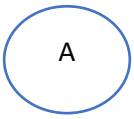
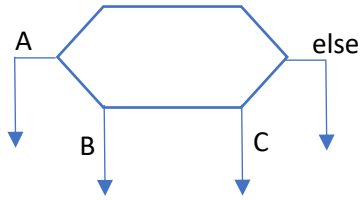
Ambas formas de expresión de la receta simbolizan el mismo algoritmo. Además, este tipo de gráficos no sólo se utilizan en el contexto de **la programación**, sino que también tiene otras utilidades en otras disciplinas como **la economía, la tecnología industrial y la psicología**.

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. **Este primero es el que se utiliza en programación**.

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.

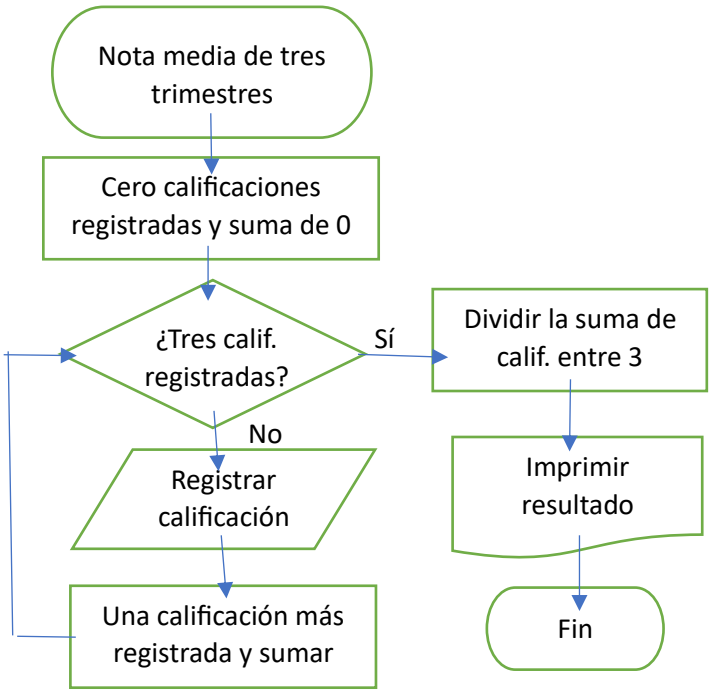
	***Bloque de proceso predefinido***. Se usará a partir del Tema 9 para indicar Subprogramas. Se explicará en su respectivo tema.
	***Conectores***. Cuando las líneas de flujo son largas o confusas, se utilizan dos pares de conectores etiquetados con letras y sustituyen a la línea.
	Sin embargo, cuando 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.

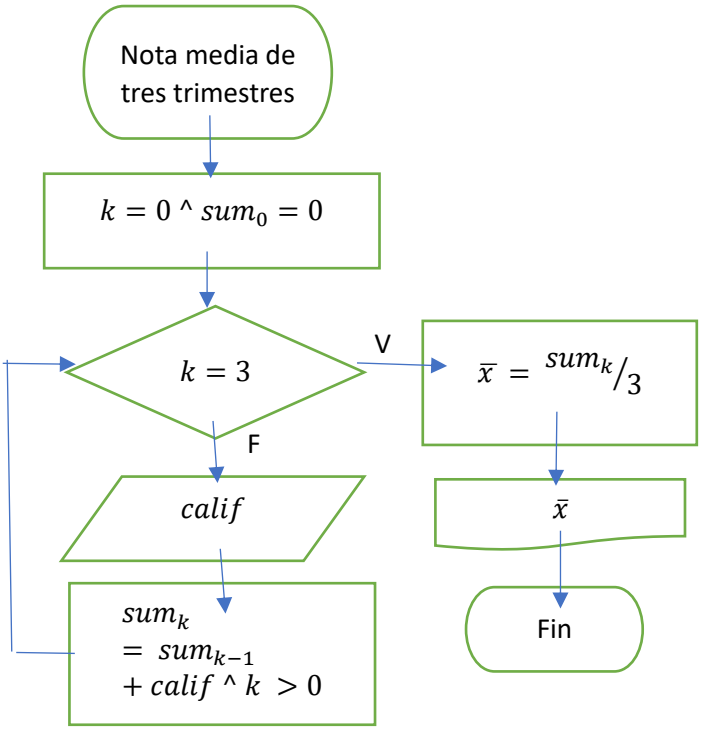
Es más, este diagrama todavía puede mejorar si recurrimos a un **lenguaje abstracto** como el **algebraico** o el **lenguaje formal de diagramas de flujo**. El lenguaje algebraico ya lo conocemos, es el que permite que en un problema como por ejemplo “La suma de edades de los tres hijos es 30. Las edades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

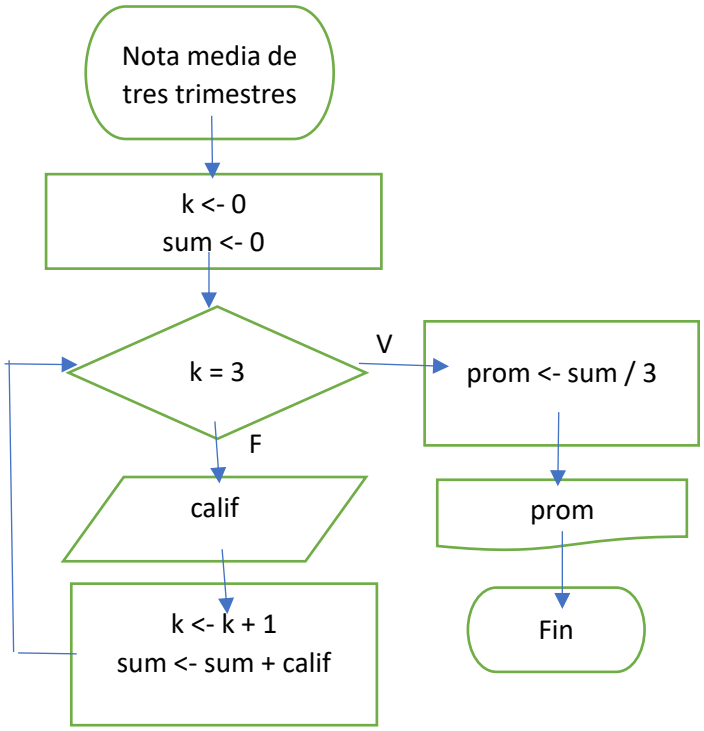
Este lenguaje **destaca por su parecido a los primeros lenguajes de programación creados, por su simplicidad y su linealidad**. Sus reglas son las siguientes:

- 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., la expresión $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$ 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:

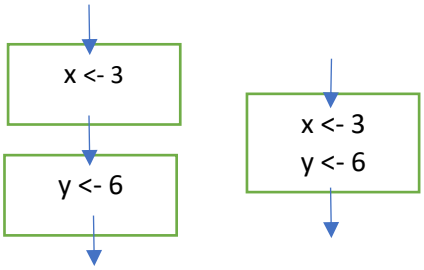


Este lenguaje formal nos expone la principal **diferencia entre la igualdad y la asignación** en el contexto de la programación, y que su concepto no es el matemático.

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.

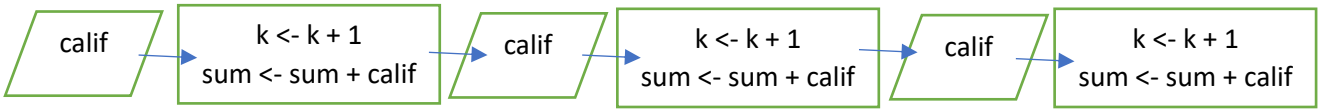
Esto permite expresiones como k <- k + 1, es decir, "asigno a la variable k el valor que tiene actualmente k más uno" Hay que tener en cuenta también que una expresión algebraica como x = x + 1 no tiene solución, por lo que por reducción al absurdo no tiene sentido hablar de igualdad en las variables,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

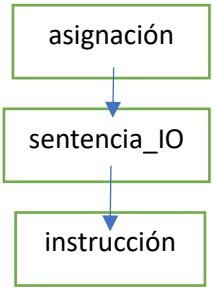
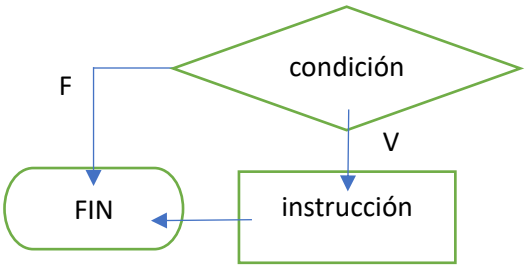
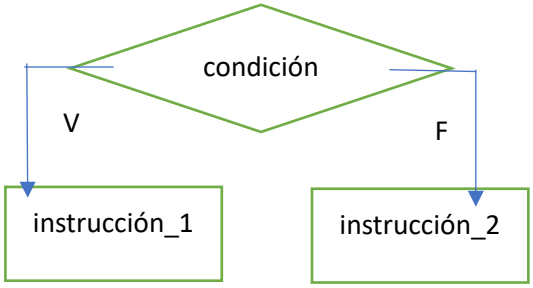
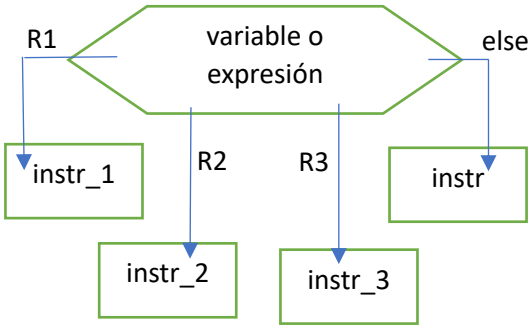
En segundo lugar, **el bucle**, que destaca por su flecha estilo [corchete, **sustituye a una sucesión de repetidas asignaciones**, ahorrando bastante espacio. En el ejemplo anterior, el bucle podría sustituirse ineficientemente por:

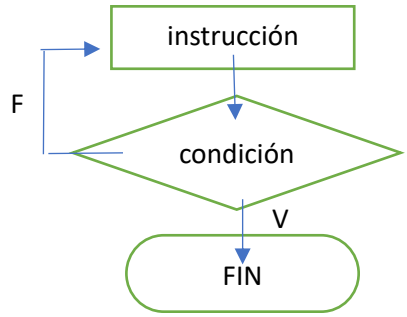
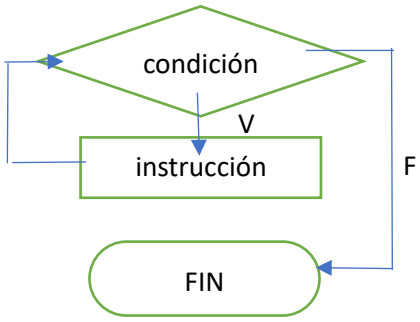
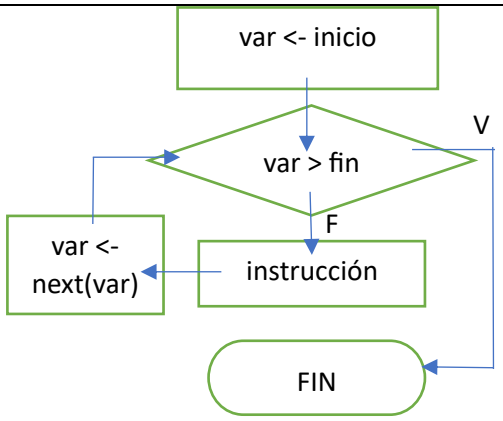


1.2 Pseudocódigo

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** típicas que podemos encontrarnos al **graficar un diagrama de flujo**. Estas estructuras corresponden con una forma peculiar de codificación o lenguaje informal llamado pseudocódigo.

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td><code><asignación>::= <variable> := <valor> |
<expresión> ;
<sentencia_IO>::= scan() | print() ;
<instrucción>::= <asignación> |
<sentencia_IO>;</code></td>
<td><code><condición>::= <expresión_booleana>;
<bloque_if>::=
if (<condición>) then
    <instrucción>; {Se cumple la cond.}</code></td>
</tr>
<tr>
<td><u>Estructura secuencial</u>.
Es aquella que ejecuta una sucesión de instrucciones secuencialmente, ejecutándolas todas y sólo una vez.</td>
<td><u>Estructura selectiva simple</u>.
Es aquella que se compone de un bloque de decisión en el cual, si y solo si la condición se cumple, se ejecuta una determinada instrucción.</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td><code><bloque_if_else>::=
if (<condición>) then
    <instrucción_1>; {Se cumple la cond}
else
    <instrucción_2>; {No se cumple}</code></td>
<td><code><bloque_cases>::=
cases of <variable> | <expresión>
    <rango_1>: <instrucción_1>;
    <rango_2>: <instrucción_2>;
    <rango_3>: <instrucción_3>;
    else: <instrucción>; {otros rangos}</code></td>
</tr>
<tr>
<td><u>Estructura selectiva doble</u>.
Es aquella que se compone de un bloque de decisión en el cual, si la condición se cumple, se ejecuta una determinada instrucción. En caso contrario, ejecuta una instrucción alternativa.</td>
<td><u>Estructura selectiva múltiple</u>.
Se compone de un bloque de condición, en el cual, dependiendo de la condición que cumpla la expresión, ejecutará una instrucción u otra. Si esta es “else”, se ejecutará en caso de que no cumpla ninguna de las condiciones anteriores.</td>
</tr>
</table>

<table>
<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:

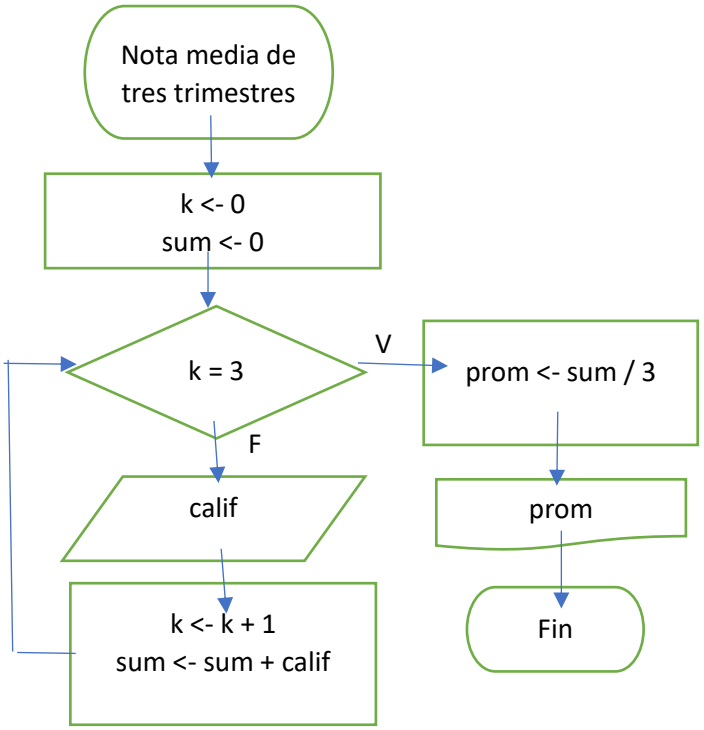
```
i:=2;
while (i<=size) do
    if (v[i-1]>v[i]) then
        aux:=v[i-1];
    end if;
end while;
v[i-1]:=v[i];
v[i]:=aux;
i:=i+1;
```

```
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.