Introducción

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Conceptos básicos

Pregunta 1

What number is the letter A in the English alphabet?

(Para ver la respuesta pulsa aquí: 1)

1.1. Informática

- Definición:

> **Informática**:
>
> La ciencia que estudia los sistemas de tratamiento automático de la información, también llamados **sistemas informáticos**.

- Estos sistemas están formados por:
  - elementos físicos (**hardware**)
  - elementos lógicos (**software**) y
  - elementos humanos (profesionales y usuarios).
- El *hardware*, a su vez, está formado por componentes:
  - Ordenadores
  - Soportes de almacenamiento
  - Redes de comunicaciones
  - ...

1.1.1. Procesamiento automático

- El procesamiento automático de la información siempre tiene el mismo esquema de funcionamiento:

- El **objetivo** del procesamiento automático de la información es **convertir los *datos de entrada* en *datos de salida*** mediante un *hardware* que ejecuta las instrucciones definidas por un *software* (**programas**).
- Los programas gobiernan el funcionamiento del *hardware*, indicándole qué tiene que hacer y cómo.
- La **Programación** es la ciencia y el arte de diseñar dichos programas.

1.1.1.1. Ejemplos

- Calcular la suma de cinco números:
  - **Entrada**: los cinco números.
  - **Proceso**: sumar cada número con el siguiente hasta acumular el resultado final.
  - **Salida**: la suma calculada.
- Dada una lista de alumnos con sus calificaciones finales, obtener otra lista ordenada de mayor a menor por la calificación obtenida y que muestre sólo los alumnos aprobados:
  - **Entrada**: Una lista de pares *(Nombre alumno, Calificación)*.
  - **Proceso**: Eliminar de la lista los pares que tengan una calificación menor que cinco y ordenar la lista resultante de mayor a menor según la calificación.

- **Salida**: la lista ordenada de alumnos aprobados.

1.2. Ordenador

1.2.1. Definición

Ordenador:

Un ordenador es una máquina que procesa información automáticamente de acuerdo con un programa almacenado.

1. Es una *máquina*.
2. Su función es *procesar información*.
3. El procesamiento se realiza de forma *automática*.
4. El procesamiento se realiza siguiendo un *programa* (*software*).
5. Este programa está *almacenado* en una memoria interna del mismo ordenador (arquitectura de **Von Neumann**).

1.2.2. Funcionamiento básico

1.2.2.1. Elementos funcionales

- Un ordenador consta de tres componentes principales:
  1. **Unidad central de proceso (CPU) o procesador**
     - *Unidad aritmético-lógica (ALU)*
     - *Unidad de control (UC)*
  2. **Memoria**
     - *Memoria principal* o central
       - *Memoria de acceso aleatorio (RAM)*
       - *Memoria de sólo lectura (ROM)*
     - *Memoria secundaria* o externa
  3. **Dispositivos de E/S**
     - *Dispositivos de entrada*
     - *Dispositivos de salida*

1.2.2.2. Unidad central de proceso (CPU) o procesador

- **Unidad aritmético-lógica (ALU)**:

  Realiza los cálculos y el procesamiento numérico y lógico.

- **Unidad de control (UC)**:

  Ejecuta de las instrucciones enviando las señales a las distintas unidades funcionales involucradas.

1.2.2.3. Memoria

- **Memoria principal** o central:

  Almacena los datos y los programas que los manipulan.

  Ambos (datos y programas) deben estar en la memoria principal para que la CPU pueda acceder a ellos.

  Dos tipos:

  - **Memoria de acceso aleatorio (RAM)**:

    Su contenido se borra al apagar el ordenador.

  - **Memoria de sólo lectura (ROM)**:

    Información permanente (ni se borra ni se puede cambiar).

    Contiene la información esencial (datos y software) para que el ordenador pueda arrancar.

- **Memoria secundaria** o externa:

  La información no se pierde al apagar el ordenador.

  Más lenta que la memoria principal, pero de mucha más capacidad.

1.2.2.4. Dispositivos de E/S

- **Dispositivos de entrada**:

  Introducen datos en el ordenador (*ejemplos*: teclado, ratón, escáner…)

- **Dispositivos de salida**:

  Vuelcan datos fuera del ordenador (*ejemplos*: pantalla, impresora…)

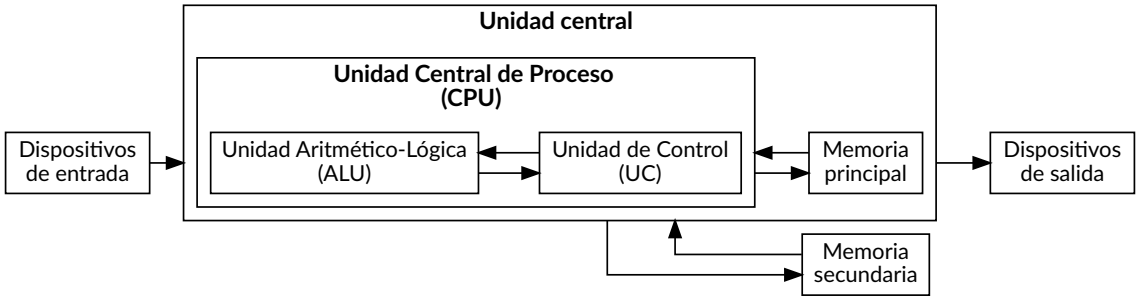
- **Dispositivos de entrada/salida**:

  Actúan simultáneamente como dispositivos de entrada y de salida (*ejemplos*: pantalla táctil, adaptador de red…)

- Los dispositivos que acceden a **soportes de almacenamiento masivo** (las **memorias secundarias**) también se pueden considerar dispositivos de E/S:

  - Los soportes de **sólo lectura** se leen con dispositivos de entrada (*ejemplo*: discos ópticos).

- Los soportes de **lectura/escritura** operan como dispositivos de entrada/salida (*ejemplos*: discos duros, pendrives, tarjetas SD...).

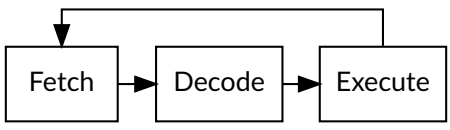


Esquema básico de un ordenador

- El programa se **carga** de la memoria secundaria a la memoria principal.
- Una vez allí, la CPU va **extrayendo** las instrucciones que forman el programa y las va **ejecutando** paso a paso, en un bucle continuo que se denomina **ciclo de instrucción**.
- Durante la ejecución del programa, la CPU recogerá los datos de entrada desde los dispositivos de entrada y los almacenará en la memoria principal, para que las instrucciones puedan operar con ellos.
- Al finalizar el programa, los datos de salida se volcarán hacia los dispositivos de salida.

1.2.2.5. Ciclo de instrucción

- En la **arquitectura Von Neumann**, los programas se almacenan en la memoria principal junto con los datos (por eso también se denomina «arquitectura de **programa almacenado**»).
- Una vez que el programa está cargado en memoria, la CPU repite siempre los mismos pasos:
  1. (**Fetch**) Busca la siguiente instrucción en la memoria principal.
  2. (**Decode**) Decodifica la instrucción (identifica qué instrucción es y se prepara para su ejecución).
  3. (**Execute**) Ejecuta la instrucción (envía las señales de control necesarias a las distintas unidades funcionales).



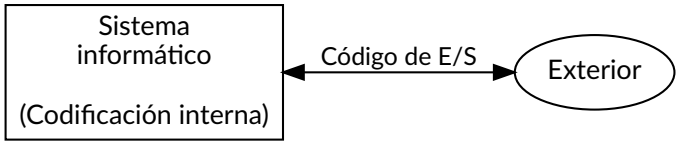
Ciclo de instrucción

1.2.2.6. Representación de información

- En un sistema informático, toda la información se almacena y se manipula en forma de números.
- Por tanto, para que un sistema informático pueda procesar información, primero hay que representar dicha información usando números, proceso que se denomina **codificación**.

> **Codificación**:
>
> Proceso mediante el cual se representa información dentro de un sistema informático, asociando a cada dato (elemental o estructurado) uno o más valores numéricos.

- Una codificación, por tanto, es una correspondencia entre un conjunto de datos y un conjunto de números llamado **código**. Al codificar, lo que hacemos es asociar a cada dato un determinado número dentro del código.
- Hay muchos tipos de información (textos, sonidos, imágenes, valores numéricos...) y eso hace que pueda haber muchas formas de codificación.
- Incluso un mismo tipo de dato (un número entero, por ejemplo) puede tener distintas codificaciones, cada una con sus características y propiedades.
- Distinguimos la forma en la que se representa la información *internamente* en el sistema informático (**codificación interna**) de la que usamos para comunicar dicha información *desde y hacia el exterior* (**codificación externa** o **de E/S**).



1.2.2.7. Codificación interna

- Los ordenadores son **sistemas electrónicos digitales** que trabajan conmutando entre varios posibles estados de una determinada magnitud física (voltaje, intensidad de corriente, etc.).
- Lo más sencillo y práctico es usar únicamente dos estados posibles.

  Por ejemplo:

  - 0 V y 5 V de voltaje.
  - 0 mA y 100 mA de intensidad de corriente.
- A cada uno de los dos posibles estados le hacemos corresponder (arbitrariamente) un valor numérico **0** ó **1**. A ese valor se le denomina **bit** (contracción de *binary digit*, dígito binario).
- Por ejemplo, la memoria principal de un ordenador está formada por millones de celdas, parecidas a microscópicos condensadores. Cada uno de estos condensadores puede estar cargado o descargado y, por tanto, es capaz de almacenar un bit:
  - Condensador cargado: bit a 1
  - Condensador descargado: bit a 0

> **Bit**:
>
> Un bit es, por tanto, la unidad mínima de información que es capaz de almacenar y procesar un ordenador, y equivale a un **dígito binario**.

- En la práctica, se usan unidades múltiplos del bit:
  - 1 byte = 8 bits
  - 1 Kibibyte (KiB) = 2^{10} = 1024 bytes
  - 1 Mebibyte (MiB) = 2^{20} bytes = 1024 Kilobytes
  - 1 Gibibyte (GiB) = 2^{30} bytes = 1024 Mebibytes
  - 1 Tebibyte (TiB) = 2^{40} bytes = 1024 Gibibytes

1.2.2.8. Sistema binario

- El sistema de numeración que usamos habitualmente los seres humanos es el **decimal** o sistema **en base diez**.
- En ese sistema disponemos de diez dígitos distintos (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9) y cada dígito en un determinado número tiene un peso que es múltiplo de una potencia de diez.

  Por ejemplo:
  $$243 = 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0$$
- El sistema de numeración que usan los ordenadores es el **sistema binario** o sistema **en base dos**, en el cual disponemos sólo de dos dígitos (0 y 1) y cada peso es múltiplo de una potencia de dos.

  Por ejemplo:
  $$101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
- Generalmente, los números naturales se codifican internamente mediante su representación en binario.
- Los números enteros se suelen codificar mediante:
  - Bit de signo (signo y magnitud)
  - Complemento a uno
  - Complemento a dos
- Los números reales se pueden codificar mediante:
  - Coma fija
  - Coma flotante
    - Simple precisión
    - Doble precisión
  - Decimal codificado en binario (BCD)

1.2.2.9. Codificación externa

- Para representar cadenas de caracteres y comunicarse con el exterior, el ordenador utiliza **códigos de E/S** o **códigos externos**.
- A cada carácter (letra, dígito, signo de puntuación, símbolo especial…) le corresponde un código (número) dentro de un **conjunto de caracteres**.
- Existen conjuntos de caracteres:
  - De **longitud fija**: a todos los caracteres les corresponden un código de igual longitud.
  - De **longitud variable**: en el mismo conjunto de caracteres hay códigos más largos y más cortos (por tanto, hay caracteres que ocupan más bytes que otros).

1.2.2.10. ASCII

- *American Standard Code for Information Interchange*.
- El conjunto de caracteres ASCII (o **código ASCII**) es el más implantado en el *hardware* de los equipos informáticos.
- Es la base de otros códigos más modernos, como el ISO-8859-1 o el Unicode.
- Es un código de 7 bits:
  - Cada carácter ocupa 7 bits.
  - Hay $2^7 = 128$ caracteres posibles.
  - Los 32 primeros códigos (del 0 al 31) son no imprimibles (códigos de control).
- El ISO-8859-1 es un código de 8 bits que extiende el ASCII con un bit más para contener caracteres latinos.

1.2.2.11. Unicode

- Con 8 bits (y con 7 bits aún menos) no es posible representar todos los posibles caracteres de todos los sistemas de escritura usados en el mundo.
- Unicode es el estándar de codificación de caracteres más completo y universal en la actualidad.
- Cada carácter en Unicode se define mediante un identificador numérico llamado *code point*.
- Unicode define tres formas de codificación:
  - **UTF-8**: codificación de 8 bits, de longitud variable (cada *code point* puede ocupar de 1 a 4 bytes). **El más usado en la actualidad.**
  - **UTF-16**: codificación de 16 bits, de longitud variable (cada *code point* puede ocupar 1 ó 2 palabras de 16 bits).
  - **UTF-32**: codificación de 32 bits, de longitud fija (cada *code point* ocupa 1 palabra de 32 bits).

1.3. Algoritmo

1.3.1. Definición

Algoritmo:

Un algoritmo es un método para resolver un problema.

- Está formado por una secuencia de pasos o **instrucciones** que se deben seguir (o **ejecutar**) para resolver el problema.
- La palabra «algoritmo» proviene de **Mohammed Al-Khowârizmi**, matemático persa que vivió durante el siglo IX y reconocido por definir una serie de reglas paso a paso para sumar, restar, multiplicar y dividir números decimales.
- **Euclides**, el gran matemático griego (del siglo IV a. C.) que inventó un método para encontrar el máximo común divisor de dos números, se considera con Al-Khowârizmi el otro gran padre de la Algorítmica (la ciencia que estudia los algoritmos).
- El estudio de los algoritmos es importante porque la resolución de un problema exige el diseño de un algoritmo que lo resuelva.
- Una vez diseñado el algoritmo, se traduce a un programa informático usando un *lenguaje de programación*.
- Finalmente, un ordenador ejecuta dicho programa.

Resolución de un problema

1.3.2. Características

- Un algoritmo debe ser:
  - **Preciso**: debe indicar el orden de ejecución de cada paso.
  - **Definido**: si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
  - **Finito**: debe terminar en algún momento, es decir, debe tener un número finito de pasos.

1.3.3. Representación

- Un algoritmo se puede describir usando el **lenguaje natural**, es decir, cualquier idioma humano.

- ¿Qué problema tiene esta forma de representación?

Ambigüedad

- En ciertos contextos la ambigüedad es asumible, pero **NO** cuando el destinatario es un ordenador.

Instrucciones para hacer una tortilla:

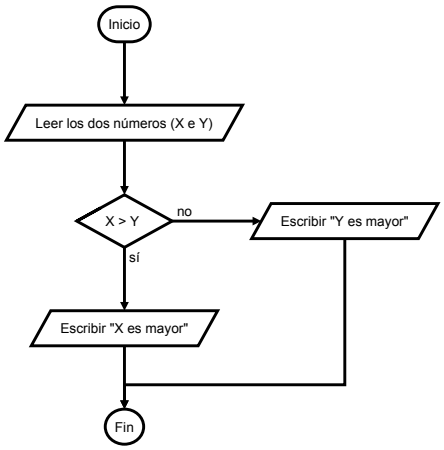
1. Coger dos huevos.
2. Encender el fuego.
3. Echar aceite a la sartén.
4. Batir los huevos.
5. Echar los huevos batidos en la sartén.
6. Esperar a que se haga por debajo.
7. Dar la vuelta a la tortilla.
8. Esperar de nuevo.
9. Sacar cuando esté lista.

Fin

1.3.3.1. Ordinograma

- Representación gráfica que describe un algoritmo en forma de diagrama de flujo.
- Las flechas indican el orden de ejecución de las instrucciones.
- Los nodos condicionales (los rombos) indican que la ejecución se bifurca a uno u otro camino dependiendo de una condición.

1.3.3.2. Ejemplo Determinar cuál es el máximo de dos números



1.3.3.3. Pseudocódigo

- Es un lenguaje semi-formal, a medio camino entre el lenguaje natural y el lenguaje que entendería un ordenador (lenguaje de programación).
- Está pensado para ser interpretado por una persona y no por un ordenador.
- En general, no se tienen en cuenta las limitaciones impuestas por el *hardware* (CPU, memoria...) o el *software* (tamaño máximo de los datos, codificación interna...), siempre y cuando no sea importante el estudio de la eficiencia o la complejidad del algoritmo.
- En ese sentido, se usa como un lenguaje de programación *idealizado*.

1.3.3.4. Ejemplo

Algoritmo: Obtener el mayor de dos números

$X \leftarrow$ **leer** número
$Y \leftarrow$ **leer** número
si $X > Y$ **entonces**
    escribir "X es mayor que Y"
si no
    escribir "Y es mayor que X"
fin

1.3.4. Cualidades deseables

- **Corrección**: El algoritmo debe solucionar correctamente el problema.
- **Claridad**: Cuanto más legible y comprensible para el ser humano, mejor.
- **Generalidad**: Un algoritmo debe resolver problemas generales. Por ejemplo, un algoritmo que sume dos números enteros debe servir para sumar cualquier pareja de números enteros, y no, solamente, para sumar dos números determinados, como pueden ser el 3 y el 5.
- **Eficiencia**: La ejecución del programa resultante de codificar un algoritmo deberá consumir lo menos posible los recursos disponibles del ordenador (memoria, tiempo de CPU, etc.).
- **Sencillez**: Hay que intentar que la solución sea sencilla, aun a costa de perder un poco de eficiencia, es decir, se tiene que buscar un equilibrio entre la claridad y la eficiencia.
- **Modularidad**: Un algoritmo puede formar parte de la solución a un problema mayor. A su vez, dicho algoritmo puede descomponerse en otros si esto favorece a la claridad del mismo.

1.3.5. Computabilidad

- ¿Todos los problemas pueden resolverse de forma algorítmica?
- Dicho de otra forma, queremos saber lo siguiente:

  Dado un problema, ¿existe un algoritmo que lo resuelva?

- Todo problema P lleva asociada una función $f_P : D \rightarrow R$, donde:

  - D es el conjunto de los datos de entrada.
  - R es el conjunto de los resultados del problema.
- Asimismo, todo algoritmo A lleva asociada una función f_A.
- La pregunta es: ¿existe un algoritmo A tal que $f_A = f_P$?
- Y de ahí vamos a la pregunta general:

> ¿Toda función f es computable (resoluble algorítmicamente)?

La respuesta es que **NO**

- Se puede demostrar que hay más funciones que algoritmos, por lo que **existen funciones que no se pueden computar mediante un algoritmo** (no son computables).
- La dificultad que tiene estudiar la computabilidad de funciones está en que no tenemos una definición formal de «*algoritmo*».
- A comienzos del S. XX, se crearon (independientemente uno del otro) dos formalismos matemáticos para representar el concepto de *algoritmo*:
  - Alonzo Church creó el **cálculo lambda**.
  - Alan Turing creó la **máquina de Turing**.
- Posteriormente se demostró que los dos formalismos eran totalmente equivalentes y eran, además, equivalentes a las **gramáticas formales**.
- Esto llevó a formular la llamada **tesis de Church-Turing**, que dice que

> «*Todo algoritmo es equivalente a una máquina de Turing.*»

- La tesis de Church-Turing es indemostrable pero prácticamente toda la comunidad científica la acepta como verdadera.
- Usando esos formalismos, se pudo demostrar que hay que problemas que no se pueden resolver mediante algoritmos.
- Uno de los problemas que no tienen una solución algorítmica es el llamado **problema de la parada**:

> **Problema de la parada**:
>
> Dado un algoritmo y un posible dato de entrada, determinar (a priori, es decir, sin ejecutarlo previamente) si el algoritmo se detendrá y producirá un valor de salida.

- Nunca podremos hacer un algoritmo que resuelva el problema de la parada en términos generales (en casos particulares sí se puede).

1.3.6. Corrección

- ¿Cómo sabemos si un algoritmo es **correcto**?
- ¿Qué significa eso de que un algoritmo sea correcto?

- Supongamos que, para un problema P, existe un algoritmo A. Lo que tenemos que averiguar es si se cumple:
$$f_P = f_A$$
- ¿Cómo lo hacemos?
  - Si el conjunto D de datos de entrada es **finito**, podríamos comparar todos los resultados de salida con los resultados esperados y ver si coinciden.
  - Si D es **infinito**, es imposible realizar una comprobación empírica de la corrección del algoritmo (en cambio, sí se puede demostrar que es incorrecto.)
- Lo mejor es recurrir a **métodos formales**:
  - **Diseño a priori**: se construye el algoritmo en base a una prueba (lo que se denomina también **demostración constructiva**).
  - **Diseño a posteriori**: se construye el algoritmo de forma más o menos intuitiva y, una vez diseñado, tratar de demostrar su corrección.
- En ambos casos, es importante definir con mucha precisión qué problema queremos resolver: se describe el problema mediante una **especificación formal**.

1.3.7. Complejidad

- ¿Cómo de **eficiente** es un algoritmo?
- La eficiencia de un algoritmo se mide en función del **consumo de recursos** que necesita el algoritmo para su ejecución.
  - Los principales recursos son el **tiempo** y el **espacio**.
- Dados dos algoritmos distintos que resuelvan el mismo problema, en general nos interesará usar el más eficiente de ellos (al margen de otras consideraciones, como la claridad, la legibilidad, la mantenibilidad, la reusabilidad, etc.)

¿Cómo medimos la eficiencia de un algoritmo?

¿Cómo comparamos la eficiencia de dos algoritmos?

- El **análisis de algoritmos** estudia la eficiencia de un algoritmo desde un punto de vista abstracto (independiente de la máquina, el lenguaje de programación, la carga de trabajo, etc.).
- Define el consumo de recursos como una función del tamaño del problema.
  - Ejemplo: $t(n) \simeq 3n^2$
- Finalmente, clasifica dicho consumo según una **notación asintótica**.
  - Ejemplo: $t(n) \in O(n^2)$

1.4. Programa

- Definición:

Programa:

Un programa es la codificación de un algoritmo en un lenguaje de programación.

- Si el algoritmo está adecuadamente definido, *traducir* ese algoritmo en un programa equivalente puede resultar trivial.
- El texto del programa escrito en ese lenguaje de programación se denomina **código fuente**. Programar, al final, consiste en escribir (*codificar*) el código fuente de nuestro programa.
- El algoritmo está pensado para ser entendido por un ser humano, mientras que el programa se escribe para ser interpretado y ejecutado por un ordenador.
- Por ello, toda posible ambigüedad que pudiera quedar en el algoritmo debe eliminarse al codificarlo en forma de programa.
- Programar depende mucho de las características del lenguaje de programación elegido.
- Lo ideal es usar un lenguaje que se parezca lo más posible al *pseudolenguaje* utilizado para describir el correspondiente algoritmo.
- En un programa también hay que considerar aspectos y limitaciones que hasta ahora no habíamos tenido en cuenta:
  - El tamaño de los datos en memoria: por ejemplo, suele haber límites en cuanto a la cantidad de dígitos que puede tener un número o su precisión decimal.
  - Restricciones en los datos: mutables vs. inmutables, de tamaño fijo vs. tamaño variable, etc.
  - La semántica de las instrucciones: un símbolo usado en un algoritmo puede tener otro significado distinto en el programa, o puede que sólo pueda usarse en el programa bajo ciertas condiciones que no hace falta considerar en el algoritmo.

1.5. Lenguaje de programación

- Definición:

Lenguaje de programación:

Un lenguaje de programación es un lenguaje formal que nos permite escribir programas de forma que puedan ser comprendidos y ejecutados por un ordenador.

2. Evolución histórica

2.1. Culturas de la programación

2.2. Ingeniería del software

3. Resolución de problemas mediante programación

3.1. Análisis del problema

3.2. Especificación

3.3. Diseño del algoritmo

3.4. Codificación del algoritmo en forma de programa

4. Paradigmas de programación

4.1. Definición

Paradigma de programación:

Es un **estilo** de desarrollar programas, es decir, un **modelo** para resolver problemas computacionales.

- Cada paradigma entiende la programación desde una perspectiva diferente, partiendo de unos conceptos básicos diferentes y con unas reglas diferentes.
- Cuando diseñamos un algoritmo o escribimos un programa, lo hacemos con base en un determinado paradigma, y éste impregna por completo la forma en la que describimos la solución al problema en el que estamos trabajando.
- No existe un único paradigma de programación y cada uno tiene sus peculiaridades que lo hacen diferente.
- Cada lenguaje de programación (o pseudocódigo) se dice que *soporta* un determinado paradigma cuando con dicho lenguaje se pueden escribir algoritmos o programas según el «estilo» que impone dicho paradigma.
- Incluso existen lenguajes *multiparadigma* capaces de soportar varios paradigmas al mismo tiempo.
- Clasificación de los paradigmas de programación más importantes:
  - Imperativo
    - Estructurado
    - Orientado a objetos
  - Declarativo
    - Funcional
    - Lógico
    - De bases de datos

4.2. Imperativo

- El **paradigma imperativo** está basado en el concepto de **sentencia**. Un programa imperativo está formado por una sucesión de sentencias que se ejecutan en orden y que llevan a cabo una de estas acciones:
  - **Cambiar el estado** interno del programa, usualmente mediante la sentencia de *asignación*.
  - Cambiar el **flujo de control** del programa, haciendo que la ejecución se bifurque (*salte*) a otra parte del mismo.
- La ejecución de un programa imperativo, por tanto, consiste en una sucesión de cambios de estado controlados por mecanismos de control y que dependen del orden en el que se realizan.
- Existen varios paradigmas con las características del paradigma imperativo, por lo que podemos decir que existen varios paradigmas imperativos.

4.2.1. Estructurado

- El **paradigma estructurado** es un paradigma imperativo en el que el flujo de control del programa se define mediante las denominadas **estructuras de control**.
- Se apoya a nivel teórico en los resultados del conocido **teorema de Böhm y Jacopini**, que establece que cualquier programa se puede escribir usando solamente tres estructuras básicas:
  - Secuencia
  - Selección
  - Iteración
- Con estas tres estructuras conseguimos que los programas se puedan leer de arriba abajo como compuestos por **bloques anidados o independientes** que se leen como un todo conjunto.
- Su aparición llevó asociada la aparición de una **metodología de desarrollo** según la cual los programas se escriben por niveles de abstracción mediante refinamientos sucesivos y usando en cada nivel sólo las tres estructuras básicas.

4.2.2. Orientado a objetos

- El **paradigma orientado a objetos** se apoya en los conceptos de **objeto** y **mensaje**.
- Un programa orientado a objetos está formado por una colección de objetos que se intercambian mensajes entre sí.
- Los objetos son entidades que existen dentro del programa y que poseen un cierto **estado interno**.
- Cuando un objeto envía un mensaje a otro, el objeto receptor del mensaje reaccionará llevando a cabo alguna acción, que probablemente provocará un **cambio en su estado interno** y que, posiblemente, provocará también el envío de mensajes a otros objetos.
- La programación orientada a objetos está vista como una forma natural de entender la programación y es, con diferencia, **el paradigma más usado en la actualidad**.

4.3. Declarativo

- La **programación declarativa** engloba a una familia de paradigmas de programación de muy alto nivel.
- En programación declarativa se describe la solución a un problema con base en las propiedades que debe cumplir dicha solución y no tanto en forma de instrucciones que se deben ejecutar para resolver el problema.
- Se dice que un programa imperativo describe **cómo** resolver el problema, mientras que un programa declarativo describe **qué** forma debe tener la solución.
- Para dar forma a la solución, se utilizan formalismos abstractos matemáticos y lógicos, lo que da lugar a los dos grandes paradigmas declarativos: la **programación funcional** y **la programación lógica**.

4.3.1. Funcional

- La **programación funcional** es un paradigma de programación declarativa basado en el uso de **funciones matemáticas**.
- Tiene su origen teórico en el **cálculo lambda** de Alonzo Church (los lenguajes funcionales se pueden considerar azúcar sintáctico del cálculo lambda).
- Una función (en programación funcional) define de un cálculo a realizar a partir de unos datos de entrada, con la propiedad de que el resultado de la función sólo puede depender de dichos datos de entrada (lo que se denomina **transparencia referencial**).
- Eso significa que una función no puede tener estado interno ni su resultado puede depender del estado interno del programa. Por tanto, no existen los **efectos laterales**.
- Demostrar la corrección de un programa funcional o paralelizar su ejecución es **mucho más fácil** que con un programa imperativo.

4.3.2. Lógico

- La **programación lógica** es un paradigma de programación declarativa basado en el uso de la **lógica de predicados de primer orden**.
- Básicamente, un programa lógico es una colección de definiciones que forman un conjunto de **axiomas** en un sistema de **deducción lógica**.
- Ejecutar un programa lógico equivale a poner en marcha un mecanismo deductivo que trata de **demostrar un teorema** a partir de los axiomas.
- El ejemplo más característico de este tipo de lenguajes es **Prolog**.

4.3.3. De bases de datos

- Los sistemas de gestión de bases de datos relacionales (SGBDR) disponen de un lenguaje que permite al usuario consultar y manipular la información almacenada.

- A esos lenguajes se los denomina **lenguajes de bases de datos** o **lenguajes de consulta**.
- El lenguaje de consulta más conocido es el **SQL**.
- Los SGBDR se basan en el *modelo relacional*, que es un modelo matemático.
- SQL es, básicamente, una implementación del **álgebra relacional**.
- Con SQL, el usuario indica *qué* desea obtener y el SGBDR determina automáticamente el mejor camino para alcanzar dicho objetivo.

5. Lenguajes de programación

5.1. Definición

5.1.1. Sintaxis

5.1.1.1. Notación EBNF

5.1.2. Semántica

5.2. Evolución histórica

5.3. Clasificación

5.3.1. Por nivel

- Dependiendo del **nivel** del lenguaje de programación tenemos:
  - Lenguajes de bajo nivel
  - Lenguajes de alto nivel

5.3.1.1. Lenguajes de bajo nivel

- Lenguajes más cercanos a la máquina
- Con poca o nula capacidad de abstracción
- Se trabaja directamente con elementos propios del *hardware* del ordenador
- Atados a la arquitectura interna de la máquina para la que se programa
- Programas difíciles de escribir, depurar, mantener y portar
- Se consigue el máximo control del ordenador
- Principales ejemplos:
  - Código máquina

  - Ensamblador

5.3.1.2. Lenguajes de alto nivel

- Lenguajes más cercanos al ser humano
- Mayor capacidad de abstracción
- Independiente de la arquitectura y los detalles internos del ordenador o el sistema operativo
- Programas más fáciles de escribir, depurar, mantener y portar
- Menor control de los recursos de la máquina
- Ejemplos de lenguajes de alto nivel:
  - Fortran
  - LISP
  - COBOL
  - BASIC
  - Pascal
  - C
  - Java
  - Ruby
  - C++
  - Python
  - JavaScript
  - C#
  - PHP
  - Haskell

5.3.2. Por generación

1. **Primera generación**: Se programa directamente en *código máquina*.
2. **Segunda generación**: Aparece el *lenguaje ensamblador* como un lenguaje simbólico que se traduce a lenguaje máquina usando un *programa ensamblador*.
3. **Tercera generación**: Aparecen los *lenguajes de alto nivel* con los que se puede programar con códigos independientes de la máquina. El código fuente se traduce a código máquina usando programas específicos llamados **traductores**.
4. **Cuarta generación**: Herramientas que combinan un lenguaje de programación de alto nivel con un *software* de generación de pantallas, listados, informes, etc. orientado al desarrollo rápido de aplicaciones. Ejemplos característicos son los lenguajes de **programación visual**.
5. **Quinta generación**: Es una denominación que se usó durante un tiempo para los lenguajes de programación de muy alto nivel (funciones y lógicos) destinados principalmente a resolver problemas de **Inteligencia Artificial**, pero como término ya ha caído en desuso.

5.3.3. Por propósito

- Dependiendo del tipo de programa que podemos escribir con el lenguaje, tenemos:
  - **Lenguajes de propósito general**: Con ellos se pueden escribir programas muy diversos. No están atados a un tipo concreto de problema a resolver. Ejemplos:
    - LISP, Pascal, C, Java, Ruby, C++, Python, C#, Haskell...
  - **Lenguajes de propósito específico**: Son lenguajes mucho más especializados y destinados principalmente a resolver un tipo determinado de problema. No sirven para escribir cualquier tipo de programa pero, dentro de su ámbito de actuación, suelen funcionar mejor que los lenguajes de propósito general. Ejemplos:
    - Lenguajes de consulta a bases de datos (SQL)
    - Lenguajes de descripción de hardware (VHDL)
    - Lenguajes para desarrollo de aplicaciones de gestión (COBOL)

5.3.4. Por paradigma

- Dependiendo del paradigma de programación que soporta el lenguaje, podemos encontrar:
  - Lenguajes imperativos
  - Lenguajes funcionales
  - Lenguajes orientados a objetos
  - Lenguajes lógicos
  - Lenguajes dirigidos por eventos
  - Lenguajes multiparadigma

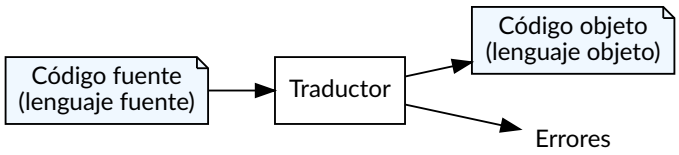
6. Traductores

6.1. Definición

- El único lenguaje que entiende la máquina directamente es el **lenguaje máquina** o **código máquina**, que es un lenguaje de **bajo nivel**.
- Para poder programar con un lenguaje de **alto nivel**, necesitamos usar herramientas *software* que traduzcan nuestro programa al lenguaje máquina que entiende el ordenador.
- Estas herramientas *software* son los **traductores**.

> **Traductor:**
>
> Es un *software* que traduce un programa escrito en un lenguaje a otro lenguaje, conservando su significado.

- El traductor transforma el programa fuente (o **código fuente**) en el programa objeto (o **código objeto**).
- El código fuente está escrito en el **lenguaje fuente** (que generalmente será un lenguaje de alto nivel).
- El código objeto está escrito en el **lenguaje objeto** (que generalmente será código máquina).
- Durante el proceso de traducción, el traductor también informa al programador de posibles **errores** en el código fuente.



El proceso de traducción

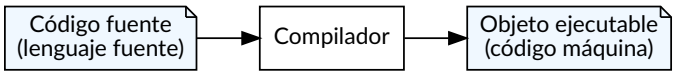
6.2. Compiladores

- Definición:

  Compilador:

  Es un traductor que convierte un programa escrito en un lenguaje de **más alto nivel** a un lenguaje de **más bajo nivel**.

- Generalmente, el lenguaje objeto suele ser **código máquina** y el resultado de la compilación es un **objeto ejecutable** directamente por la máquina.



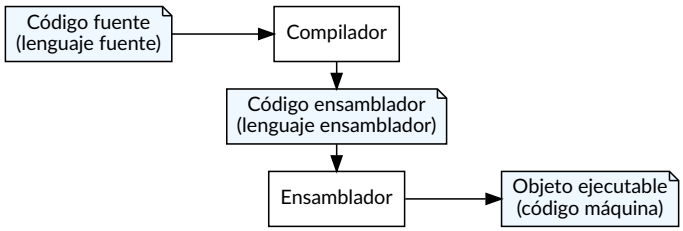
6.2.1. Ensambladores

- Un caso particular de compilador es el **ensamblador**:

  Ensamblador:

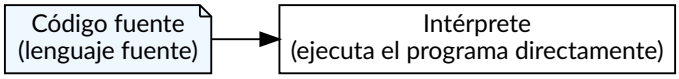
  Es un compilador que traduce un programa escrito en **lenguaje ensamblador** a código máquina.

- Muchas veces, los compiladores se construyen *en cadena*: en lugar de generar código máquina directamente, generan código ensamblador que sirve de entrada a un programa ensamblador que generará el código objeto final.



6.3. Intérpretes

- Un **intérprete** es un caso muy especial de traductor.
- En lugar de generar código objeto, el intérprete **lee el código fuente y lo ejecuta directamente**.
- El intérprete funciona, por tanto, como un **simulador** de una máquina que hablara el lenguaje de alto nivel en el que está escrito el programa fuente.



- El desarrollo de programas se ve **acelerado** con un intérprete ya que no es necesario pasar por el proceso de compilación ni, por tanto, generar el código objeto para poder ejecutar el programa.
- Sin embargo, si el programa fuente tiene errores sintácticos, el intérprete no informará de ellos hasta el momento en el que intente ejecutar la instrucción errónea, es decir, **los errores se muestran en tiempo de ejecución, no en tiempo de compilación**.
- Hay lenguajes *compilados* y lenguajes *interpretados*, e incluso lenguajes que son ambas cosas (tienen compiladores e intérpretes).

6.3.1. Interactivos (*REPL*)

- A los intérpretes que hemos visto hasta ahora se les denomina **intérpretes por lotes**, ya que tratan al programa fuente como un lote de instrucciones conjuntas.
- A diferencia de los anteriores, los **intérpretes interactivos** son programas que solicitan al programador que introduzca por teclado, una a una, las instrucciones que se desean ejecutar, y el intérprete las va ejecutando a medida que el programador las va introduciendo.
- Su comportamiento se resume en el siguiente bucle:
  1. Leer la siguiente instrucción por teclado (***R**ead*)
  2. Ejecutar o evaluar la instrucción (***E**val*)
  3. Imprimir por la pantalla el resultado (***P**rint*)
  4. Repetir el bucle (***L**oop*)
- Los intérpretes interactivos son ideales para:
  - Aprender conceptos de programación.

- Experimentar con el lenguaje.
- Probar rápidamente el efecto de una instrucción.

7. Entornos integrados de desarrollo

7.1. Terminal

7.1.1. `Zsh`

7.1.2. Oh My Zsh

7.1.3. `less`

7.2. Editores de texto

7.2.1. Editores vs. IDE

7.2.2. Vim y less

7.2.3. Visual Studio Code

7.2.3.1. Instalación

7.2.3.2. Configuración

7.2.3.3. Extensiones

Respuestas a las preguntas

Respuesta a la Pregunta 1

- 1

¡The letter A is the first letter in the alphabet!

Bibliografía

Joyanes Aguilar, Luis. 2008. *Fundamentos de Programación*. Aravaca: McGraw-Hill Interamericana de España.