1.1 Introducción

1. Conceptos Básicos

1.1 Computadora

Una **computadora**, o como se la conoce popularmente, un **ordenador**, es una máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información, capaz de resolver problemas matemáticos y lógicos mediante la utilización automática de programas informáticos.

1.2 Informática

Conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores.

1.3 Hardware

Componentes físicos que forman parte de un ordenador (o de otro dispositivo electrónico): procesador, RAM, impresora, teclado, ratón,...

1.4 Software

Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación.

1.5 Sistema Operativo

Se trata del software encargado de gestionar el ordenador. Es la aplicación que oculta la física real del ordenador para mostrarnos una interfaz que permita al usuario un mejor y más fácil manejo de la computadora. Por ejemplo, : Windows, Linux, MacOS.

1.6 Algoritmo

Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

1.7 Programa Informático

Es una secuencia de instrucciones escritas para realizar una tarea específica en una computadora.

1.8 Aplicación Informática

Software formado por uno o más programas, la documentación de los mismos y los archivos necesarios para su funcionamiento, de modo que el conjunto completo forma una herramienta de trabajo en un ordenador.

2. Codificación de la información

Un ordenador maneja información de todo tipo. Nuestra perspectiva humana nos permite rápidamente diferenciar lo que son números, de lo que es texto, imagen,...Sin embargo al tratarse de una máquina digital, el ordenador sólo es capaz de representar números en forma binaria. Por ello todos los ordenadores necesitan codificar la información del mundo real al equivalente binario entendible por el ordenador.

2.1 Sistemas Numéricos

Existen dos tipos de sistemas numéricos:

1. **Sistemas no posicionales**

   En ellos se utilizan símbolos cuyo valor numérico es siempre el mismo independientemente de donde se sitúen. Es lo que ocurre con la numeración romana. En esta numeración el símbolo I significa siempre uno independientemente de su posición.

2. **Sistemas posicionales**

   En ellos los símbolos numéricos cambian de valor en función de la posición que ocupen. Es el caso de nuestra numeración, el símbolo 2, en la cifra 12 vale 2; mientras que en la cifra 21 vale veinte.

La historia ha demostrado que los sistemas posicionales son mucho mejores para los cálculos matemáticos ya que las operaciones matemáticas son más sencillas. Todos los sistemas posicionales tienen una **base**, que es el número total de símbolos que utiliza el sistema.

2.1.1 Sistemas posicionales

- Sistema decimal: la base es 10 ya que utiliza 10 símbolos, desde el 0 hasta el 9.
- Sistema binario: la base es 2, utiliza 0 y 1.
- Sistema octal: la base es 8, desde el 0 hasta el 7.
- Sistema hexadecimal: la base es 16, desde el 0 al 9 y desde la A a la F.

Para convertir un número octal en binario, se representa cada dígito en octal por tres dígitos binarios según la siguiente tabla de conversión:

Octal	Binario
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Por lo tanto, el número 467 en octal sería 100110111 en binario.

Lo mismo podemos hacer con el binario y el hexadecimal pero con 4 dígitos.

Hexadecimal	Binario
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110

Hexadecimal	Binario
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

El número B3F en hexadecimal es 101100111111 en binario.

2.2 Representación de texto en el sistema binario

Puesto que una computadora no sólo maneja números, habrá dígitos binarios que contengan información que no es traducible a decimal. Todo depende de cómo se interprete esa traducción. Por ejemplo en el caso del texto, lo que se hace es codificar cada carácter en una serie de números binarios. El código **ASCII** es un estándar que ha sido durante mucho tiempo el más utilizado. Inicialmente era un código que utilizaba 7 bits para representar texto, lo que significaba que era capaz de codificar 127 caracteres. Por ejemplo el número 65 (1000001 en binario) se utiliza para la A mayúscula. Poco después apareció un problema: este código es suficiente para los caracteres del inglés, pero no para otras lenguas. Entonces se añadió el octavo bit para representar otros 128 caracteres que son distintos según idiomas(Europa Occidental usa unos códigos que no utiliza Europa Oriental). Eso provoca que un código como el 190 signifique cosas diferentes si cambiamos de país. Por ello cuando un ordenador necesita mostrar texto, tiene que saber qué juego de códigos debe de utilizar (lo cual supone un tremendo problema). Una ampliación de este método de codificación de caracteres es el estándar **Unicode** que puede utilizar hasta 4 bytes (32 bits) con lo que es capaz de codificar cualquier carácter en cualquier lengua del planeta utilizando el mismo conjunto de códigos. Poco a poco es el código que se va extendiendo, siendo actualmente utilizado en un número considerable de tecnologías recientes, como XML, Java y sistemas operativos modernos.

La descripción completa del estándar está disponible en la página web de Unicode https://unicode.org/. En *Quick Links → Code Charts* encontraremos las tablas de caracteres. Los caracteres básicos del español los encontraremos en *Latin → Basic Latin (ASCII)* y los caracteres especiales del español como por ejemplo, las vocales acentuadas y la ñ, en *Latin1 → Supplement*.

2.3 Representación binaria de datos no numéricos ni de texto

En el caso de datos más complejos (imágenes, vídeo, audio) se necesita una codificación más compleja. En el caso, por ejemplo de las imágenes, una forma básica de codificarlas en binario es la que graba cada píxel (cada punto distinguible en la imagen) mediante tres bytes: el primero graba el nivel de rojo, el segundo el nivel de azul y el tercero el nivel de verde. Y así por cada píxel. Esto se conoce como modelo de color RGB

donde es posible representar un color mediante la mezcla por adición de los tres colores de luz primarios. Por ejemplo un punto en una imagen de color rojo puro: `11111111 00000000 00000000` Naturalmente en una imagen no solo se graban los píxeles sino el tamaño de la imagen, el modelo de color,... de ahí que representar estos datos sea tan complejo para el ordenador (y tan complejo entenderlo para nosotros).

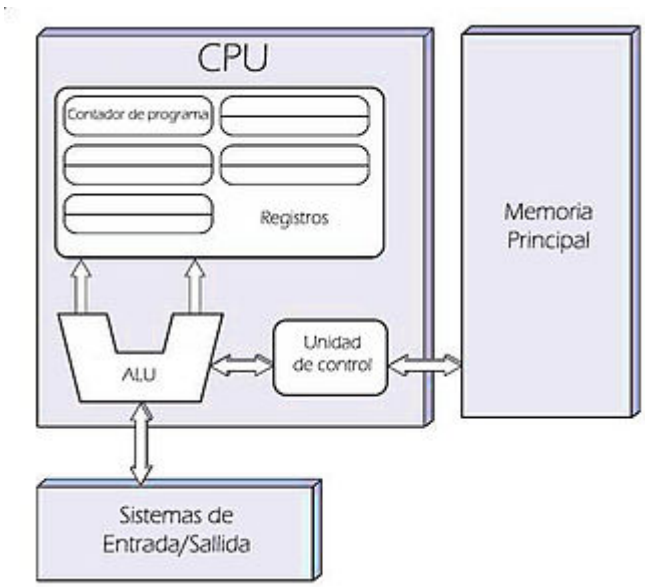
2.4 Múltiplos para medir dígitos binarios

Puesto que toda la información de un ordenador se representa de forma binaria, se hizo indispensable el utilizar unidades de medida para poder indicar la capacidad de los dispositivos:

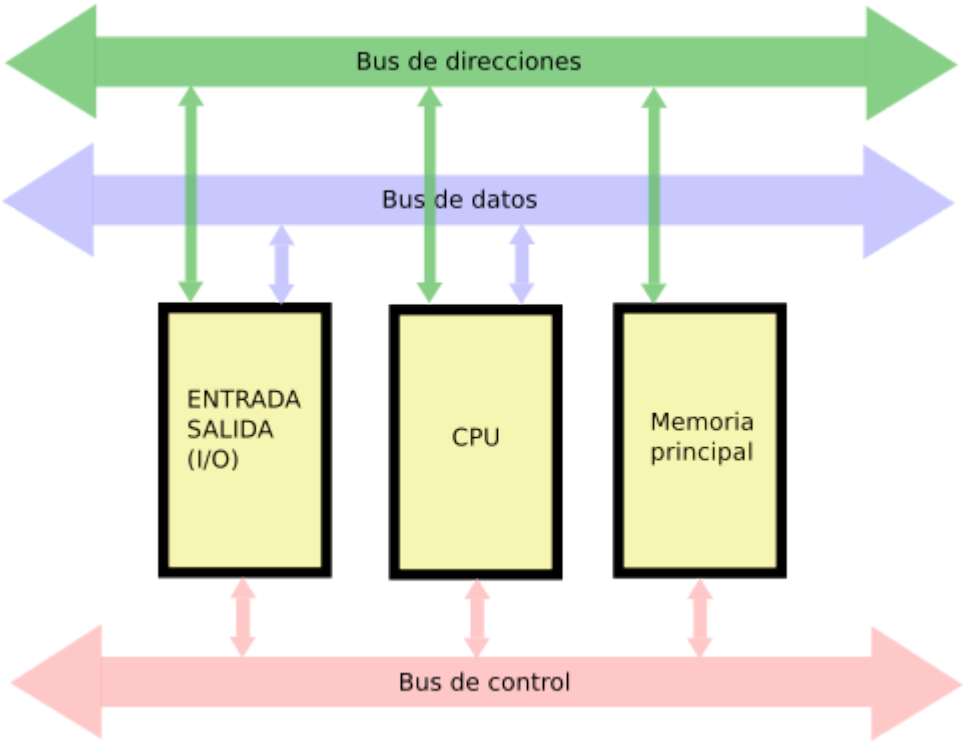
- **Bit** (de binary digit). Representa un dígito binario. Por ejemplo, se dice que el número binario 1001 tiene cuatro bits.
- **Byte**. Es el conjunto de 8 bits.
- **Kilobyte**. Son 1024 bytes.
- **Megabyte**. Son 1024 Kilobytes.
- **Gigabyte**. Son 1024 Megabytes.
- **Terabyte**. Son 1024 Gigabytes.
- **Petabyte**. Son 1024 Terabytes.

3. Arquitectura de Von Newmann

La mayoría de los sistemas informáticos actuales se basan en la arquitectura propuesta por Von Newmann. Esta arquitectura se caracteriza porque el programa que ejecuta el sistema informático está almacenado internamente en el propio sistema.



Los buses transportan la información entre los diferentes elementos.



Bus de datos:

Como su nombre indica transporta datos. Estos datos pueden ser la información que se está procesando o las instrucciones del programa que se ejecuta. El ancho en bits del bus de datos define el tamaño de la *palabra* del sistema informático, por ejemplo, 32 bits ó 64 bits.

Bus de direcciones:

El bus de direcciones se utiliza para indicar el origen y/o el destino de los datos. En el bus de direcciones se indica la *posición de memoria* a la que se está accediendo en cada momento. Puede tratarse de una dirección de la memoria principal o puede tratarse de una dirección de memoria de un periférico. El ancho en bits del bus de direcciones determina el tamaño del espacio de memoria direccionable. Un ancho de 16 bits puede almacenar 2 elevado a 16 (65.536) valores diferentes.

Bus de control:

El bus de control proporciona señales para coordinar las diferentes tareas que se realizan en el sistema informático. Por ejemplo, *R/W* indica si es una operación de lectura o escritura.

4. Historia del Software

Los **primeros ordenadores** cumplían una única programación que estaba definida en los componentes eléctricos que formaban el ordenador. La idea de que el ordenador hiciera varias tareas (ordenador programable o multipropósito) hizo que se idearan las tarjetas perforadas. En ellas se utilizaba código binario, de modo que se hacían agujeros en ellas para indicar el código 1 o el cero. Estos “primeros programas” lógicamente servían para hacer tareas muy concretas.

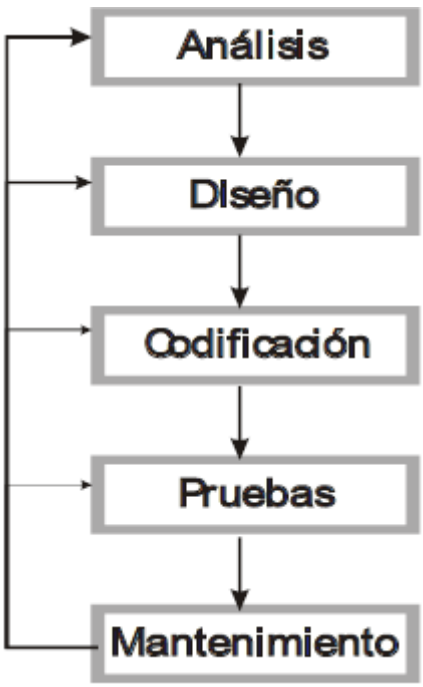
La llegada de **ordenadores electrónicos** más potentes hizo que los ordenadores se convirtieran en verdaderas máquinas digitales que seguían utilizando el 1 y el 0 del código binario pero que eran capaces de leer miles de unos y ceros. Empezaron a aparecer los primeros lenguajes de programación que escribían código más entendible por los humanos que posteriormente era convertido al código entendible por la

máquina.

Inicialmente la **creación de aplicaciones** requería escribir pocas líneas de código en el ordenador, por lo que no había una técnica especificar a la hora de crear programas. Cada programador se defendía como podía generando el código a medida que se le ocurría. Poco a poco las funciones que se requerían a los programas fueron aumentando produciendo miles de líneas de código que al estar desorganizada hacían casi imposible su mantenimiento. Sólo el programador que había escrito el código era capaz de entenderlo y eso no era en absoluto práctico.

La llamada **crisis del software** ocurrió cuando se percibió que se gastaba más tiempo en hacer las modificaciones a los programas que en volver a crear el software. La razón era que ya se habían codificado millones de líneas de código antes de que se definiera un buen método para crear los programas. La solución a esta crisis ha sido la definición de la **Ingeniería del software** como un oficio que requería un método de trabajo similar al del resto de ingenierías. La búsqueda de una metodología de trabajo que elimine esta crisis parece que aún no está resuelta, de hecho los métodos de trabajo siguen redefiniéndose una y otra vez.

5. Ciclo de vida de una aplicación



Una de las cosas que se han definido tras el nacimiento de la ingeniería del software ha sido el ciclo de vida de una aplicación. El ciclo de vida define los pasos que sigue el proceso de creación de una aplicación desde que se propone hasta que finaliza su construcción. Los pasos son:

1. **Análisis**. En esta fase se determinan los requisitos que tiene que cumplir la aplicación. Se anota todo aquello que afecta al futuro funcionamiento de la aplicación. Este paso le realiza un analista.
2. **Diseño**. Se especifican los esquemas de diseño de la aplicación. Estos esquemas forman los planos del programador, los realiza el analista y representan todos los aspectos que requiere la creación de la aplicación.
3. **Codificación**. En esta fase se pasa el diseño a código escrito en algún lenguaje de programación. Esta es la primera labor que realiza el programador.
4. **Pruebas**. Se trata de comprobar que el funcionamiento de la aplicación es la adecuada. Se realiza en varias fases:

a) **Prueba del código**. Las realizan programadores. Normalmente programadores distintos a los que crearon el código, de ese modo la prueba es más independiente y generará resultados más óptimos.

b) **Versión alfa**. Es una primera versión terminada que se revisa a fin de encontrar errores. Estas pruebas conviene que sean hechas por personal no informático. El producto sólo tiene cierta apariencia de acabado.

c) **Versión beta**. Versión casi definitiva del software en la que no se estiman fallos, pero que se distribuye a los clientes para que encuentren posibles problemas. A veces está versión acaba siendo la definitiva.

5. **Mantenimiento**. Tiene lugar una vez que la aplicación ha sido ya distribuida. En esta fase se asegura que el sistema siga funcionando aunque cambien los requisitos o el sistema para el que fue diseñado el software. Antes esos cambios se hacen los arreglos pertinentes, por lo que habrá que retroceder a fases anteriores del ciclo de vida.

6. Errores

Cuando un programa obtiene una salida que no es la esperada, se dice que posee errores. Los errores son uno de los caballos de batalla de los programadores ya que a veces son muy difíciles de encontrar (de ahí que hoy en día en muchas aplicaciones se distribuyan parches para subsanar errores no encontrados en la creación de la aplicación). Tipos de errores:

- **Error del usuario**. Errores que se producen cuando el usuario realiza algo inesperado y el programa no reacciona apropiadamente (se entiende por usuario la persona que utiliza la aplicación informática).
- **Errores de documentación**. Ocurren cuando la documentación del programa no es correcta y provoca fallos en el manejo.
- **Error de interfaz**. Se entiende por interfaz el medio con que el usuario se comunica con la máquina, como ventanas, menús, etc. El error de interfaz ocurre si la interfaz de usuario de la aplicación es enrevesada para el usuario impidiendo su manejo normal. También se llaman así los errores de protocolo entre dispositivos.
- **Error de entrada / salida o de comunicaciones**. Ocurre cuando falla la comunicación entre el programa y un dispositivo (se desea imprimir y no hay papel, falla el teclado,...)
- **Error fatal**. Ocurre cuando el hardware produce una situación inesperada que el software no puede controlar (el ordenador se cuelga, errores en la grabación de datos,...)
- **Error de sintaxis**. Ocurre cuando una instrucción del código no está bien escrita, es decir, tiene un error de sintaxis. Por lo tanto, no puede ser traducida a código binario.
- **Error de ejecución**. Se produce cuando el ordenador no puede ejecutar alguna instrucción de forma correcta. Por ejemplo, la instrucción `c = 5 / 0;` es correcta sintácticamente y será traducida a código binario. Sin embargo, cuando la computadora intente realizar la división 5 / 0 se producirá un error de ejecución, ya que, matemáticamente, no se puede dividir entre cero.
- **Error de lógica**. En cuanto a los errores de lógica son los más difíciles de detectar. Cuando un programa no tiene errores de sintaxis ni de ejecución, pero aún así, no funciona bien, esto es debido a la existencia de algún error lógico. De manera que, un error de lógica se produce cuando los resultados obtenidos no son los esperados.

7. Lenguajes de Programación

7.1 Historia

Inicios de la programación

Charles Babbage definió a mediados del siglo XIX lo que él llamó la máquina analítica. Se considera a esta máquina el diseño del primer ordenador. La realidad es que no se pudo construir hasta el siglo siguiente. El caso es que su colaboradora Ada Lovelace escribió en tarjetas perforadas una serie de instrucciones que la máquina iba a ser capaz de ejecutar. Se dice que eso significó el inicio de la ciencia de la programación de ordenadores. En la segunda guerra mundial debido a las necesidades militares, la ciencia de la computación prospera y con ella aparece el famoso ENIAC (Electronic Numerical Integrator And Calculator), que se programaba cambiando su circuitería. Esa es la primera forma de programar (que aún se usa en numerosas máquinas) que sólo vale para máquinas de único propósito. Si se cambia el propósito, hay que modificar la máquina.

Código máquina. Primera generación de lenguajes (1GL)

No mucho más tarde apareció la idea de que las máquinas fueran capaces de realizar más de una aplicación. Para lo cual se ideó el hecho de que hubiera una memoria donde se almacenaran esas instrucciones. Esa memoria se podía rellenar con datos procedentes del exterior. Inicialmente se utilizaron tarjetas perforadas para introducir las instrucciones. Durante mucho tiempo esa fue la forma de programar, que teniendo en cuenta que las máquinas entendían sólo código binario, consistía en introducir la programación de la máquina mediante unos y ceros, el llamado código máquina. Todavía los ordenadores es el único código que entienden, por lo que cualquier forma de programar debe de ser convertida a código máquina. Sólo se ha utilizado por los programadores en los inicios de la informática. Su incomodidad de trabajo hace que sea impensable para ser utilizado hoy en día. Pero cualquier programa de ordenador debe, finalmente, ser convertido a este código para que un ordenador puede ejecutar las instrucciones de dicho programa. Un detalle a tener en cuenta es que el código máquina es distinto para cada tipo de procesador. Lo que hace que los programas en código máquina no sean portables entre distintas máquinas.

Lenguaje ensamblador. Segunda generación de lenguajes (2GL)

En los años 40 se intentó concebir un lenguaje más simbólico que permitiera no tener que programar utilizando código máquina. Poco más tarde se ideó el lenguaje ensamblador, que es la traducción del código máquina a una forma más textual. Cada tipo de instrucción se asocia a una palabra mnemotécnica (como SUM para sumar por ejemplo), de forma que cada palabra tiene traducción directa en el código máquina. Tras escribir el programa en código ensamblador, un programa (llamado también ensamblador) se encargará de traducir el código ensamblador a código máquina. Esta traducción es rápida puesto que cada línea en ensamblador tiene equivalente directo en código máquina (en los lenguajes modernos no ocurre esto). La idea es la siguiente: si en el código máquina el número binario 0000 significa sumar, una instrucción máquina que sumara el número 8 (00001000 en binario) al número 16 (00010000 en binario) sería `0000000100000010000`. El ordenador entendería que los primeros cuatro bits representan la instrucción y los 8 siguientes el primer número y los ocho siguientes el segundo número (suponiendo que los números ocupan 8 bits). Lógicamente trabajar de esta forma es muy complicado. Por eso se podría utilizar la siguiente traducción en ensamblador: `SUM 8 16`, que ya se entiende mucho mejor. Puesto que el ensamblador es una representación textual pero exacta del código máquina, cada programa sólo funcionará para la máquina en la que fue concebido el programa, es decir, no es portable. La ventaja de este lenguaje es que se puede controlar absolutamente el funcionamiento de la máquina, lo que permite crear programas muy eficientes. Lo malo es precisamente que hay que conocer muy bien el funcionamiento de la computadora para crear programas con esta técnica. Además las líneas requeridas para realizar una tarea se disparan ya que las instrucciones de la máquina son excesivamente simples.

Lenguajes de alto nivel. Lenguajes de tercera generación (3GL)

Aunque el ensamblador significó una notable mejora sobre el código máquina, seguía siendo excesivamente críptico. De hecho para hacer un programa sencillo, se necesitaban miles y miles líneas de código. Para evitar los problemas del ensamblador apareció la tercera generación de lenguajes de programación, la de los lenguajes de alto nivel. En este caso el código vale para cualquier máquina pero deberá ser traducido mediante software especial que adaptará el código de alto nivel al código máquina correspondiente. Esta traducción es necesaria ya que el código en un lenguaje de alto nivel no se parece en absoluto al código máquina. Tras varios intentos de representar lenguajes, en 1957 aparece el que se considera el primer lenguaje de alto nivel, el **FORTRAN** (FORmula TRANslation), lenguaje orientado a resolver fórmulas matemáticas. Poco a poco fueron evolucionando los lenguajes formando lenguajes cada vez mejores. Así en 1958 se crea **LISP** como lenguaje declarativo para expresiones matemáticas, en 1960 se creó el **COBOL** como lenguaje de gestión y en 1963 se creó **PL/I** el primer lenguaje que admitía la multitarea y la programación modular. **BASIC** se creó en el año 1964 como lenguaje de programación sencillo de aprender y ha sido uno de los lenguajes más populares. En 1968 se crea **LOGO** para enseñar a programar a los niños. **Pascal** se creó con la misma idea académica pero siendo ejemplo de lenguaje estructurado para programadores avanzados. El creador del Pascal (Niklaus Wirdth) creó **Modula** en 1977 siendo un lenguaje estructurado para la programación de sistemas (intentando sustituir al C). **C** es un lenguaje de programación originalmente desarrollado por Dennis Ritchie entre 1969 y 1972 como evolución del anterior lenguaje B a su vez basado en BCPL. Al igual que B, es un lenguaje orientado a la implementación de Sistemas operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones. Es un lenguaje de medio nivel ya que dispone de las estructuras típicas de los lenguajes de alto nivel pero a su vez dispone de construcciones del lenguaje que permiten un control a muy bajo nivel pudiendo acceder directamente a memoria o dispositivos periféricos.

Lenguajes de cuarta generación (4GL)

En los años 70 se empezó a utilizar éste término para hablar de lenguajes en los que apenas hay código y en su lugar aparecen indicaciones sobre qué es lo que el programa debe de obtener. Se consideraba que el lenguaje **SQL** (muy utilizado en las bases de datos) y sus derivados eran de cuarta generación. Los lenguajes de consulta de datos, creación de formularios, informes,... son lenguajes de cuarta generación. Aparecieron con los sistemas de base de datos. Actualmente se consideran lenguajes de éste tipo a aquellos lenguajes que se programan sin escribir casi código (lenguajes visuales), mientras que también se propone que este nombre se reserve a los lenguajes orientados a objetos.

Lenguajes orientados a objetos

En los 80 llegan los lenguajes preparados para la programación orientada a objetos todos procedentes de **Simula** (1964) considerado el primer lenguaje con facilidades de uso de objetos. De estos destacó inmediatamente **C++**. A partir de C++ aparecieron numerosos lenguajes que convirtieron los lenguajes clásicos en lenguajes orientados a objetos y además con mejoras en el entorno de programación, son los llamados lenguajes visuales: **Visual Basic**, **Delphi** (versión orientada a objetos de Pascal), **Visual C++**,...En 1995 aparece **Java** como lenguaje totalmente orientado a objetos y en el año 2000 aparece **C#** un lenguaje que toma la forma de trabajar de C++ y del propio Java.

Lenguajes para la Web

La popularidad de Internet ha producido lenguajes híbridos que se mezclan con el código **HTML** con el que se crean las páginas web. HTML no es un lenguaje en sí sino un formato de texto pensado para crear páginas web. Estos lenguajes se usan para poder realizar páginas web más potentes. Son lenguajes interpretados como **JavaScript** o **VB Script**, o lenguajes especiales para uso en servidores como **ASP, JSP o PHP**. Todos

ellos permiten crear páginas web usando código mezcla de páginas web y lenguajes de programación sencillos.

7.2 Tipos de lenguajes

Según el estilo de programación se puede hacer esta división:

- **Lenguajes imperativos**. Son lenguajes que se centran en cómo resolver el problema. Las instrucciones se ejecutan secuencialmente y van modificando la memoria del ordenador para producir las salidas requeridas. La mayoría de lenguajes (C, Pascal, Basic, Cobol, ...) son de este tipo. Dentro de estos lenguajes están también los lenguajes orientados a objetos (C++, Java, C#,...).
- **Lenguajes declarativos**. Son lenguajes que se centran en el qué queremos resolver en lugar de en cómo resolverlo. El más conocido de ellos es el SQL, lenguaje de consulta de Bases de datos.
- **Lenguajes funcionales**. Definen funciones que nos responden a través de una serie de argumentos. Son lenguajes que usan expresiones matemáticas. El más conocido de ellos es el LISP.
- **Lenguajes lógicos**. Lenguajes utilizados para resolver expresiones lógicas. Utilizan la lógica para producir resultados. El más conocido es el PROLOG.

7.3 Intérpretes y compiladores

A la hora de convertir un programa en código máquina, se pueden utilizar dos tipos de software: intérpretes y compiladores.

Intérpretes

Se convierte cada línea a código máquina y se ejecuta ese código máquina antes de convertir la siguiente línea. De esa forma si las dos primeras líneas son correctas y la tercera tiene un fallo de sintaxis, veríamos el resultado de las dos primeras líneas y al llegar a la tercera se nos notificaría el fallo y finalizaría la ejecución. El intérprete hace una simulación de modo que parece que la máquina entiende directamente las instrucciones del lenguaje, pareciendo que ejecuta cada instrucción (como si fuese código máquina directo). El BASIC era un lenguaje interpretado, se traducía línea a línea. Hoy en día la mayoría de los lenguajes integrados en páginas web son interpretados, la razón es que como la descarga de Internet es lenta, es mejor que las instrucciones se vayan traduciendo según van llegando en lugar de cargar todas en el ordenador. Por eso lenguajes como JavaScript son interpretados.

Un programa que se convierte a código máquina mediante un intérprete sigue estos pasos:

1. Lee la primera instrucción
2. Comprueba si es correcta
3. Convierte esa instrucción al código máquina equivalente
4. Lee la siguiente instrucción
5. Vuelve al paso 2 hasta terminar con todas las instrucciones

Ventajas de los intérpretes:

- Se tarda menos en crear el primer código máquina. El programa se ejecuta antes.
- No hace falta cargar todas las líneas para empezar a ver resultados (lo que hace que sea una técnica idónea para programas que se cargan desde Internet)

Desventajas de los intérpretes:

- El código máquina producido es peor ya que no se optimiza al valorar una sola línea cada vez. El código optimizado permite estudiar varias líneas a la vez para producir el mejor código máquina posible, por ello no es posible mediante el uso de intérpretes.

- Todos los errores son errores en tiempo de ejecución, no se pueden detectar antes de lanzar el programa. Esto hace que la depuración de los errores sea más compleja.
- El código máquina resultante gasta más espacio.
- Hay errores difícilmente detectables, ya que para que los errores se produzcan, las líneas de errores hay que ejecutarlas. Si la línea es condicional, hasta que no probemos todas las posibilidades del programa, no sabremos todos los errores de sintaxis cometidos.

Compiladores

Se trata de software que traduce las instrucciones de un lenguaje de programación de alto nivel a código máquina. La diferencia con los intérpretes reside en que se analizan todas las líneas antes de empezar la traducción. Durante muchos años, los lenguajes potentes han sido compilados. El uso masivo de Internet ha propiciado que esta técnica a veces no sea adecuada y haya lenguajes modernos interpretados o semi-interpretados, mitad se compila hacia un código intermedio y luego se interpreta línea a línea (esta técnica la siguen Java y los lenguajes de la plataforma .NET de Microsoft).

Ventajas de los compiladores:

- Se detectan errores antes de ejecutar el programa (errores de compilación).
- El código máquina generado es más rápido (ya que se optimiza).
- Es más fácil hacer procesos de depuración de código.

Desventajas de los compiladores:

- El proceso de compilación del código es lento.
- No es útil para ejecutar programas desde Internet ya que hay que descargar todo el programa antes de traducirle, lo que ralentiza mucho su uso.

8. Tipos de paradigmas de programación

Un **paradigma de programación** es un modelo básico de diseño y desarrollo de programas que permite generar programas con un conjunto de normas específicas.

En general, la mayoría de paradigmas son variantes de los dos tipos principales de programación, imperativa y declarativa. En la programación ***imperativa*** se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema.

En la programación ***declarativa*** las sentencias que se utilizan lo que hacen es describir el problema que se quiere solucionar; se programa diciendo lo que se quiere resolver pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de deducción de información a partir de la descripción realizada.

A continuación se describen algunas de las distintas variantes de paradigmas de programación:

8.1 Programación imperativa

Es uno de los paradigmas de programación de computadoras más utilizados. Bajo este paradigma, la programación se describe en términos del estado del programa y de sentencias que cambian dicho estado. Java es un lenguaje imperativo, lo que implica que un programa Java está compuesto por una secuencia de instrucciones, que son ejecutadas en el mismo orden en el que se escriben, de manera que al ejecutarla se produce cambios en el estado del programa.

Dentro de esta categoría se engloban la programación estructurada y la programación orientada a objetos, las cuales han permitido mejorar la mantenibilidad y la calidad de los programas imperativos.

8.2 Programación estructurada

Está orientada a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a subrutinas y tres estructuras básicas: secuencia, selección (*if* y *switch*) e iteración(bucles *for* y *while*).

Una subrutina o subprograma, como idea general, se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.

8.3 Programación orientada a objetos

La programación orientada a objetos encapsula elementos denominados objetos.

Bajo este paradigma, la programación se describe como una serie de objetos independientes que se comunican entre sí. Java es un lenguaje orientado a objetos.

8.4 Programación orientada a eventos

La programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o lo que sea que esté accionando el programa.

Mientras que en la programación estructurada es el programador el que define cuál va a ser el flujo del programa, en la programación dirigida por eventos será el propio usuario, o lo que sea que esté accionando el programa, el que dirija el flujo del programa. Aunque en la programación estructurada puede haber intervención de un agente externo al programa, estas intervenciones ocurrirán cuando el programador lo haya determinado, y no en cualquier momento como puede ser en el caso de la programación dirigida por eventos.

8.5 Programación declarativa

Está basada en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla (tan solo se le indica a la computadora qué es lo que se desea obtener o qué es lo que se está buscando). Los lenguajes declarativos tienen la ventaja de ser razonados matemáticamente, lo que permite el uso de mecanismos matemáticos para optimizar el rendimiento de los programas.

Dentro de esta categoría se engloban la programación funcional y la programación lógica.

8.6 Programación funcional

Es un paradigma de programación en el que el resultado de un programa deriva de la aplicación de distintas funciones a la entrada, sin cambiar el estado interno del programa. En la programación funcional los bloques principales de construcción de nuestros programas son las funciones, y no los objetos.

Al aplicar programación funcional se produce normalmente un código más corto y más sencillo de entender que aplicando programación imperativa.

8.7 Programación lógica

Es un paradigma de programación basado en la definición de relaciones lógicas.

8.8 Programación multiparadigma

Es el uso de dos o más paradigmas dentro de un programa. Por ejemplo, Java es imperativo y orientado a objetos.