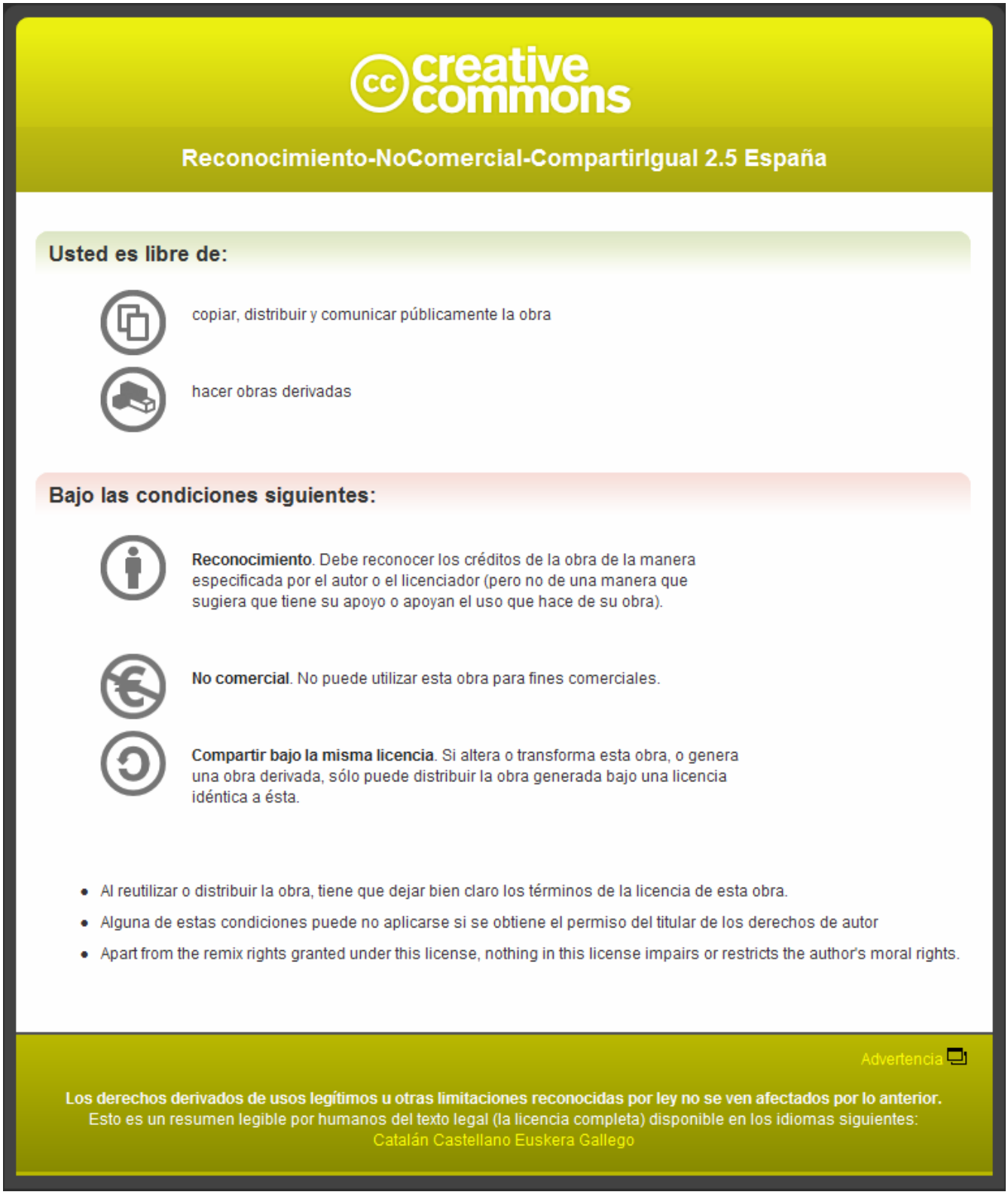
Unidad 1: Algoritmos y programas

Fundamentos de Programación. 1º de ASI



Para ver una copia completa de la licencia, acudir a la dirección http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es

(1)
algoritmos y programas

esquema de la unidad

(1.1) computadora y sistema operativo

(1.1.1) computadora

Según la **RAE** (Real Academia de la lengua española), una computadora 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***.

En cualquier caso cualquier persona tiene una imagen clara de lo que es una computadora, o como se la conoce popularmente, un **ordenador**. La importancia del ordenador en la sociedad actual es importantísima; de hecho casi no hay tarea que no esté apoyada en la actualidad por el ordenador.

Debido a la importancia y al difícil manejo de estas máquinas, aparece la **informática** como el ***conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores***.

Inicialmente, las primeras computadoras eran máquinas basadas en el funcionamiento de relés o de ruedas. Por ello sólo eran capaces de realizar una única tarea.

A finales de los años cuarenta **Von Newman** escribió en un artículo lo que serían las bases del funcionamiento de los ordenadores (seguidos en su mayor parte hasta el día de hoy).

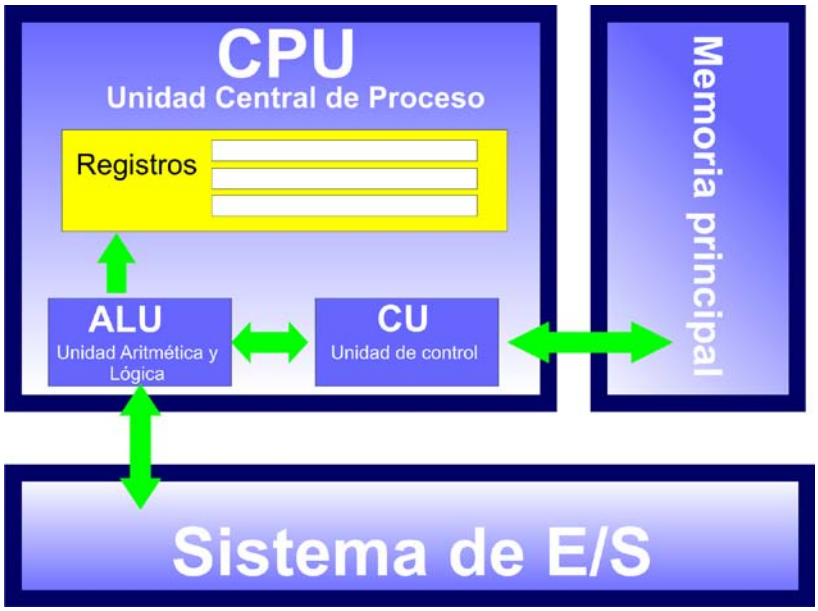


Ilustración 1, Modelo de Von Newman

Las mejoras que consiguió este modelo (entre otras) fueron:

- Incluir el modelo de **Programa Almacenado** (fundamental para que el ordenador pueda realizar más de una tarea)
- Aparece el concepto de **Lenguaje de Programación**.
- Aparece el concepto de programa como amo secuencia de instrucciones secuenciales (aunque pueden incluir bifurcaciones y saltos).

El modelo no ha cambiando excesivamente hasta la actualidad de modo que el modelo actual de los ordenadores es el que se indica en la Ilustración 2.

De los componentes internos del ordenador, cabe destacar el **procesador** (o microprocesador, muchas veces se le llama ***microprocesador*** término que hace referencia al tamaño del mismo e incluso simplemente ***micro***). Se trata de un chip que contiene todos los elementos de la **Unidad Central de Proceso**; por lo que es capaz de realizar e interpretar instrucciones. En realidad un procesador sólo es capaz de realizar tareas sencillas como:

- Operaciones aritméticas simples: suma, resta, multiplicación y división
- Operaciones de comparación entre valores
- Almacenamiento de datos

Ilustración 2, arquitectura de los ordenadores actuales

En definitiva los componentes sobre los que actualmente se hace referencia son:

- **Procesador.** Núcleo digital en el que reside la CPU del ordenador. Es la parte fundamental del ordenador, la encargada de realizar todas las tareas.
- **Placa base.** Circuito interno al que se conectan todos los componentes del ordenador, incluido el procesador.
- **Memoria RAM.** Memoria principal del ordenador, formada por un circuito digital que está conectado mediante tarjetas a la placa base. Su contenido se pierde cuando se desconecta al ordenador. Lo que se almacena no es permanente. Mientras el ordenador está funcionando contiene todos los programas y datos con los que el ordenador trabaja.
- **Memoria caché.** Memoria ultrarrápida de características similares a la RAM, pero de velocidad mucho más elevada por lo que se utiliza para almacenar los últimos datos utilizados de la memoria RAM.
- **Periféricos.** Aparatos conectados al ordenador mediante tarjetas o ranuras de expansión (también llamados puertos). Los hay de **entrada** (introducen datos en el ordenador: teclado, ratón, escáner,...), de **salida** (muestran datos desde el ordenador: pantalla, impresora, altavoces,...) e incluso de **entrada/salida** (módem, tarjeta de red).
- **Unidades de almacenamiento.** En realidad son periféricos, pero que sirven para almacenar de forma permanente los datos que se deseen del ordenador. Los principales son el **disco duro** (unidad de gran tamaño interna al ordenador), la **disquetera** (unidad de baja capacidad y muy lenta, ya en desuso), el **CD-ROM** y el **DVD.**

(1.1.2) hardware y software

hardware

Se trata de todos los componentes físicos que forman parte de un ordenador (o de otro dispositivo electrónico): procesador, RAM, impresora, teclado, ratón,...

software

Se trata de la parte conceptual del ordenador. Es decir los datos y aplicaciones que maneja. De forma más práctica se puede definir como cualquier cosa que se pueda almacenar en una unidad de almacenamiento es software (la propia unidad sería hardware).

(1.1.3) Sistema Operativo

Se trata del software (programa) encargado de gestionar el ordenador. Es la aplicación que oculta la física real del ordenador para mostrarnos un interfaz que permita al usuario un mejor y más fácil manejo de la computadora.

funciones del Sistema Operativo

Las principales funciones que desempeña un Sistema Operativo son:

- Permitir al usuario comunicarse con el ordenador. A través de comandos o a través de una interfaz gráfica.
- Coordinar y manipular el hardware de la computadora: memoria, impresoras, unidades de disco, el teclado,...
- Proporcionar herramientas para organizar los datos de manera lógica (carpetas, archivos,...)
- Proporcionar herramientas para organizar las aplicaciones instaladas.
- Gestionar el acceso a redes
- Gestionar los errores de hardware y la pérdida de datos.
- Servir de base para la creación de aplicaciones, proporcionando funciones que faciliten la tarea a los programadores.
- Administrar la configuración de los usuarios.
- Proporcionar herramientas para controlar la seguridad del sistema.

algunos sistemas operativos

- **Windows.** A día de hoy el Sistema Operativo más popular (instalado en el 95% de computadoras del mundo). Es un software propiedad de Microsoft por el que hay que pagar por cada licencia de uso.

- **Unix.** Sistema operativo muy robusto para gestionar redes de todos los tamaños. Actualmente en desuso debido al uso de Linux (que está basado en Unix), aunque sigue siendo muy utilizado para gestionar grandes redes (el soporte sigue siendo una de las razones para que se siga utilizando)
- **Solaris.** Versión de Unix para sistemas de la empresa **Sun**.
- **MacOs.** Sistema operativo de los ordenadores **MacIntosh**. Muy similar al sistema Windows y orientado al uso de aplicaciones de diseño gráfico.

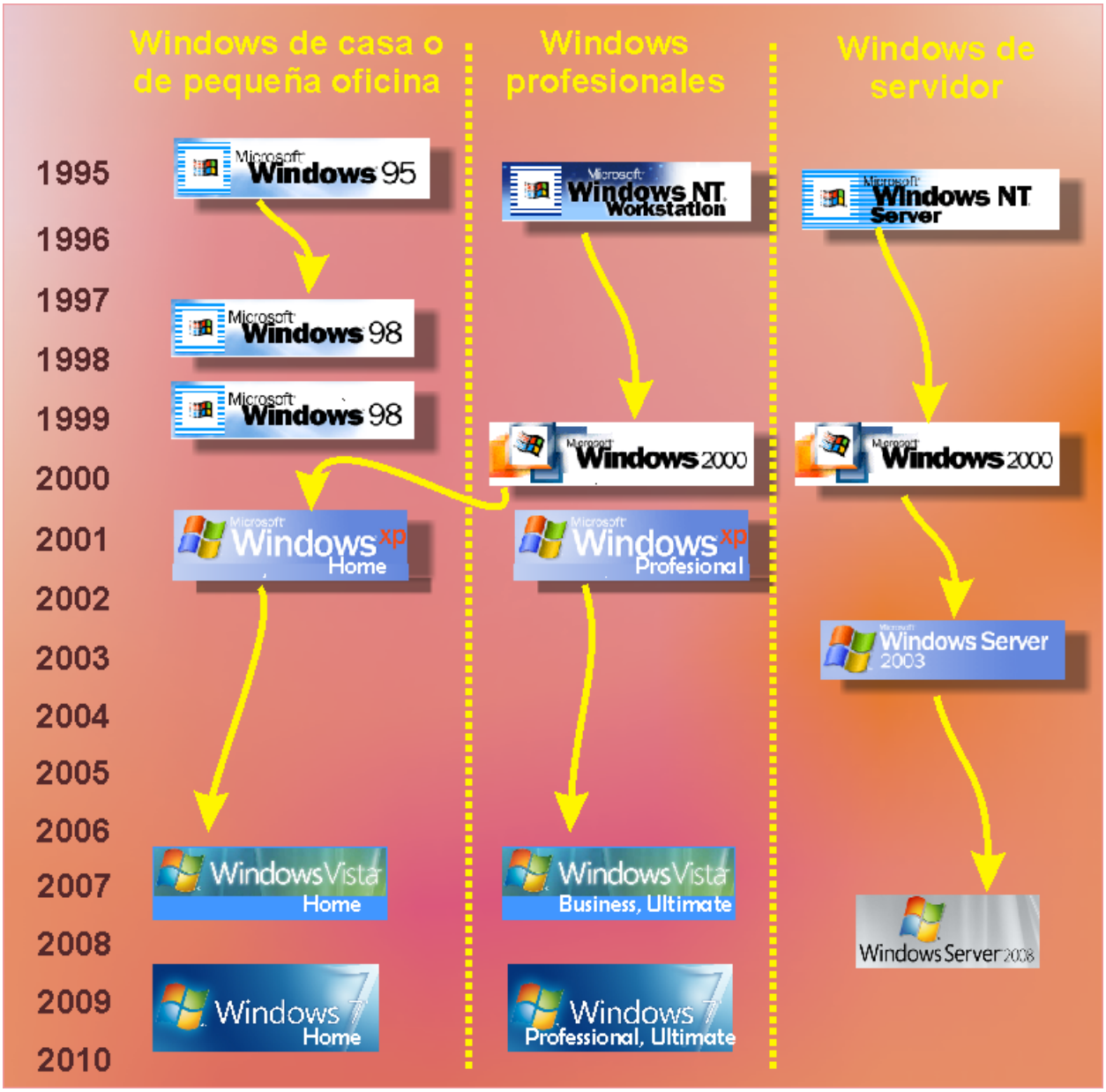


Ilustración 3, Histórico de versiones de Windows

- **Linux.** Sistema operativo de código abierto, lo que significa que el código fuente está a disposición de cualquier programador, lo que permite adecuar el sistema a las necesidades de cada usuario.

  Esta libertad ha hecho que posea numerosas distribuciones, muchas de ellas gratuitas. La variedad de distribuciones y opciones complica su aprendizaje al usuario inicial, pero aumenta las posibilidades de selección de un sistema adecuado.

  La sintaxis de Linux está basada en Linux, de hecho se trata de un Unix de código abierto pensado fundamentalmente para los ordenadores de tipo PC.

  Actualmente las distribuciones Linux más conocidas son:

- **Red Hat**
- **Fedora** (versión gratuita de Red Hat)
- **Debian**
- **Ubuntu** (variante de Debian de libre distribución, quizá el Linux más exitoso de la actualidad)
- **Mandriva**
- **SUSE**

(1.2) codificación de la información

(1.2.1) introducción

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 a el equivalente binario entendible por el ordenador.

Desde los inicios de la informática la codificación ha sido problemática y por la falta de acuerdo en la representación. Pero hoy día ya tenemos numerosos estándares.

Fundamentalmente la información que un ordenador maneja es: Números y Texto. Cualquier otro tipo de información (imagen, sonido, vídeo,…) se considera binaria (aunque como ya hemos comentado, toda la información que maneja un ordenador es binaria).

(1.2.2) sistemas numéricos

En general, a lo largo de la historia han existido numerosos sistemas de numeración. Cada cultura o civilización se ha servido en la antigüedad de los sistemas que ha considerado más pertinentes. Para simplificar, dividiremos a todos los sistemas en dos tipos:

- **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.
- **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 por lo que han retirado a los no posicionales. La razón: las operaciones matemáticas son más sencillas utilizando sistemas posicionales.

Todos los sistemas posicionales tienen una **base**, que es el número total de símbolos que utiliza el sistema. En el caso de la numeración decimal la base es 10; en el sistema binario es 2.

El **Teorema Fundamental de la Numeración** permite saber el valor decimal que tiene cualquier número en cualquier base. Dicho teorema utiliza la fórmula:

$$\ldots + X_3 \cdot B^3 + X_2 \cdot B^2 + X_1 \cdot B^1 + X_0 \cdot B^0 + X_{-1} \cdot B^{-1} + X_{-2} \cdot B^{-2} + \ldots$$

Donde:

- X_i Es el símbolo que se encuentra en la posición número i del número que se está convirtiendo. Teniendo en cuenta que la posición de las unidades es la posición 0 (la posición -1 sería la del primer decimal)
- **B** Es la base del sistemas que se utiliza para representar al número

Por ejemplo si tenemos el número ***153,6*** utilizando e sistema octal (base ocho), el paso a decimal se haría:

$$1 \cdot 8^2 + 5 \cdot 8^1 + 3 \cdot 8^0 + 6 \cdot 8^{-1} = 64 + 40 + 3 + 6/8 = 107{,}75$$

(1.2.3) sistema binario de numeración

conversión binario a decimal

El **teorema fundamental de la numeración** se puede aplicar para saber el número decimal representado por un número escrito en binario. Así para el número binario 10011011011 la conversión se haría:

$$1 \cdot 2^{10} + 0 \cdot 2^9 + 0 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1243$$

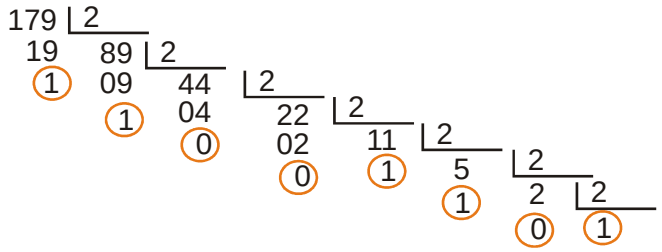
Puesto que simplemente es sumar las potencias de los dígitos con el número 1, se puede convertir asignando un peso a cada cifra. De modo que la primera cifra (la que está más a la derecha) se corresponde con la potencia 2^0 o sea uno, la siguiente 2^1 o sea dos, la siguiente 2^2 que es cuatro y así sucesivamente; de esta forma, por ejemplo para el número 10100101, se haría:

Pesos:	128	64	32	16	8	4	2	1
	1	**0**	**1**	**0**	**0**	**1**	**0**	**1**

ahora se suman los pesos que tienen un 1: 128 + 32 + 4 +1=165

conversión decimal a binario

El método más utilizado es ir haciendo divisiones sucesivas entre dos. Consiste en dividir el número decimal continuamente entre 2 tomando los restos a la inversa y el último cociente. Por ejemplo para pasar el 179:



Ahora las cifras binarias se toman al revés. Con lo cual, el número 10110011 es el equivalente en binario de 179.

Otra posibilidad más rápida es mediante restas sucesivas. En este caso se colocan todas las potencias de dos hasta sobrepasar el número decimal. La primera potencia (el primer peso) que es menor a nuestro número será un 1 en el número binario. Se resta nuestro número menos esa potencia y se coge el resultado. Se ponen a 0 todas las potencias mayores que ese resultado y un 1 en la primera potencia menor. Se vuelve a restar y así sucesivamente. Ejemplo:

Número decimal: 179

1) Obtener potencias de 2 hasta sobrepasar el número:

1 2 4 8 16 32 64 128 256

2) Proceder a las restas (observar de abajo a arriba):

Potencias	Pesos	Restas
1	1	1-1=0
2	1	3-2=1
4	0	
8	0	
16	1	19-16=3
32	1	51-32 =19
64	0	
128	1	179-128 =51
256	0	

2) El número binario es el 10110011

representación binaria de números negativos

Si se representan números negativos, entonces se debe dedicar un bit para el signo del número. En el caso de usar un byte para el número, el primer bit es el signo y los otros siete el valor absoluto.

Así:

0 1 1 1 0 1 0 1 es el número 117
1 1 1 1 0 1 0 1 es el número -117

El rango representable con 1 bit sería del -127 al 127. Hay otra posibilidad y es la representación en **complemento a uno**. En esta representación, a los números negativos se les invierten todos los bits. Ejemplo:

1 0 0 0 1 0 1 0 es el número -117 en complemento a uno

El problema en ambos casos es que hay dos representaciones para el número cero (la 00000000 y la 10000000) y por ello es más popular el **complemento a dos**. Esta forma consiste en sumar uno al complemento a uno. De este modo se añade al rango el número -128 que se representa como 10000000. Ejemplo de complemento a dos:

1 0 0 0 1 0 1 1 es el número -117 en complemento a dos

operaciones con números binarios

suma

La suma se efectúa igual que con los decimales, sólo que usando binarios según esta tabla:

Suma	Resultado
0 + 0	0
0 + 1	1
1 + 0	1
1 + 1	0 (más un acarreo de 1)

Ejemplo:

```
          1  1  1          acarreos
   1  0  1  0  1  0  1  0
+  0  1  0  0  1  1  1  1
-------------------------
   1  1  1  1  1  0  0  1
```

resta

Funciona como la resta normal, los acarreos se ponen en el segundo número.

Suma	Resultado
0 - 0	0
0 - 1	1 (más un acarreo de 1)
1 - 0	1
1 - 1	0

Ejemplo:

```
  1 0 1 0 1 0 1 0
- 0 1 0 0 1 1 1 1      acarreos
  ---------------
  0 1 0 1 1 0 1 1
```

Otra posibilidad es pasar el sustraendo a complemento a dos o a complemento a uno y entonces efectuar la suma. Eso es lo que suelen hacer casi todos los procesadores.

lógicas

Los bits se pueden manipular con operaciones lógicas. Fundamentalmente con **AND** y **OR**:

AND	Resultado
0 AND 0	0
0 AND 1	0
1 AND 0	0
1 AND 1	1

OR	Resultado
0 OR 0	1
0 OR 1	1
1 OR 0	1
1 OR 1	0

Esas operaciones se pueden realizar con números completos. Ejemplo:

```
AND 1 0 1 0 1 0 1 0
    0 1 0 0 1 1 1 1
    ---------------
    0 0 0 0 1 0 1 0
```

Ejemplo de uso de OR:

```
OR  1 0 1 0 1 0 1 0
    0 1 0 0 1 1 1 1
    ---------------
    1 1 1 0 1 1 1 1
```

Otra operación es **NOT** que trabaja con un solo número al que le invierte las cifras:

NOT	Resultado
NOT 0	1
NOT 1	0

Con números largos:

```
NOT  0 1 0 0 1 1 1 1
     ---------------
     1 0 1 1 0 0 0 0
```

sistema octal

Un caso especial de numeración es el sistema octal. En este sistema se utiliza base ocho, es decir cifras de 0 a 7. Así el número ***8*** en decimal, pasa a escribirse ***10*** en octal.

Lo interesante de este sistema es que tiene una traducción directa y rápida al sistema binario, ya que una cifra octal se corresponde exactamente con tres cifras binarias (tres bits). La razón es que el número ocho es 2^3. De ahí que tres bits signifique una cifra octal. Los números del 0 al 7 en octal pasados a binario quedarían así:

Decimal	Binario (tres bits)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

De este modo el número binario **1100011**, se agrupa de tres en tres (empezando por la derecha), es decir 1 100 011 y en octal quedaría: **143.** Del mismo modo, el número octal **217** en binario es el **010 001 111**

sistema hexadecimal

Se utiliza muchísimo en informática por razones parecidas al sistema octal. Es el sistema de numeración en base 16. Tiene la ventaja de que una cifra hexadecimal representa exactamente 4 bits (lo que comúnmente se conoce como un **nibble**). De este modo 2 cifras hexadecimales representan un byte (8 bits), que es la unidad básica para medir.

Puesto que en nuestro sistema no disponemos de 16 cifras, sino solo de 10. se utilizan letras para representar los números del 10 al 15. Tabla:

Hexadecimal	Decimal	Binario (cuatro bits)
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000

Hexadecimal	Decimal	Binario (cuatro bits)
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Así el número binario **10110011** se agruparía como 1011 0011 y sería el **B3** en hexadecimal. A su vez el número hexadecimal **CA** quedaría **1100 1010**

(1.2.4) 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** 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 es el código **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; pero la preponderancia histórica que ha tenido el código ASCII, complica su popularidad.

(1.2.5) 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. Además en estos datos no hay estándares, por lo que hay decenas de formas de codificar.

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.

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).

(1.2.6) 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 que manejaban dichos números.

Así tenemos:

- **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. Es importante porque normalmente cada carácter de texto almacenado en un ordenador consta de 8 bits (luego podemos entender que en 20 bytes nos caben 20 caracteres).
- **Kilobyte**. Son 1024 bytes.
- **Megabyte**. Son 1024 Kilobytes.
- **Gigabyte**. Son 1024 Megabytes.
- **Terabyte**. Son 1024 Gigabytes.
- **Petabyte**. Son 1024 Terabytes.

(1.3) algoritmos

(1.3.1) noción de algoritmo

Según la **RAE**: ***conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.***

Los algoritmos, como indica su definición oficial, son una serie de pasos que permiten obtener la solución a un problema. La palabra algoritmo procede del matemático Árabe **Mohamed Ibn Al Kow Rizmi**, el cual escribió sobre los años 800 y 825 su obra ***Quitad Al Mugabala***, donde se recogía el sistema de numeración hindú y el concepto del cero. **Fibonacci**, tradujo la obra al latín y la llamó: ***Algoritmi Dicit***.

El lenguaje algorítmico es aquel que implementa una solución teórica a un problema indicando las operaciones a realizar y el orden en el que se deben efectuarse. Por ejemplo en el caso de que nos encontremos en casa con una bombilla fundida en una lámpara, un posible algoritmo sería:

(1) Comprobar si hay bombillas de repuesto

(2) En el caso de que las haya, sustituir la bombilla anterior por la nueva

(3) Si no hay bombillas de repuesto, bajar a comprar una nueva a la tienda y sustituir la vieja por la nueva

Los algoritmos son la base de la programación de ordenadores, ya que los **programas de ordenador** se puede entender que son algoritmos escritos en un código especial entendible por un ordenador.

Lo malo del diseño de algoritmos está en que no podemos escribir lo que deseemos, el lenguaje ha utilizar no debe dejar posibilidad de duda, debe recoger todas las posibilidades.

Por lo que los tres pasos anteriores pueden ser mucho más largos:

Comprobar si hay bombillas de repuesto
[1.1] Abrir el cajón de las bombillas
[1.2] Observar si hay bombillas

Si hay bombillas:
[1.3] Coger la bombilla
[1.4] Coger una silla
[1.5] Subirse a la silla
[1.6] Poner la bombilla en la lámpara

Si no hay bombillas
[1.7] Abrir la puerta
[1.8] Bajar las escaleras....

Cómo se observa en un algoritmo las instrucciones pueden ser más largas de lo que parecen, por lo que hay que determinar qué instrucciones se pueden utilizar y qué instrucciones no se pueden utilizar. En el caso de los algoritmos preparados para el ordenador, se pueden utilizar sólo instrucciones muy concretas.

(1.3.2) características de los algoritmos

características que deben de cumplir los algoritmos obligatoriamente

- **Un algoritmo debe resolver el problema para el que fue formulado**. Lógicamente no sirve un algoritmo que no resuelve ese problema. En el caso de los programadores, a veces crean algoritmos que resuelven problemas diferentes al planteado.
- **Los algoritmos son independientes del ordenador**. Los algoritmos se escriben para poder ser utilizados en cualquier máquina.
- **Los algoritmos deben de ser precisos**. Los resultados de los cálculos deben de ser exactos, de manera rigurosa. No es válido un algoritmo que sólo aproxime la solución.
- **Los algoritmos deben de ser finitos**. Deben de finalizar en algún momento. No es un algoritmo válido aquel que produce situaciones en las que el algoritmo no termina.
- **Los algoritmos deben de poder repetirse**. Deben de permitir su ejecución las veces que haga falta. No son válidos los que tras ejecutarse una vez, ya no pueden volver a hacerlo por la razón que sea.

características aconsejables para los algoritmos

- **Validez.** Un algoritmo es válido si carece de errores. Un algoritmo puede resolver el problema para el que se planteó y sin embargo no ser válido debido a que posee errores
- **Eficiencia.** Un algoritmo es eficiente si obtiene la solución al problema en poco tiempo. No lo es si es lento en obtener el resultado.
- **Óptimo.** Un algoritmo es óptimo si es el más eficiente posible y no contiene errores. La búsqueda de este algoritmo es el objetivo prioritario del programador. No siempre podemos garantizar que el algoritmo hallado es el óptimo, a veces sí.

(1.3.3) elementos que conforman un algoritmo

- **Entrada.** Los datos iniciales que posee el algoritmo antes de ejecutarse.
- **Proceso.** Acciones que lleva a cabo el algoritmo.
- **Salida**. Datos que obtiene finalmente el algoritmo.

(1.4) aplicaciones

(1.4.1) programas y aplicaciones

- **Programa.** La definición de la **RAE** es: ***Conjunto unitario de instrucciones que permite a un ordenador realizar funciones diversas, como el tratamiento de textos, el diseño de gráficos, la resolución de problemas matemáticos, el manejo de bancos de datos***, etc. Pero normalmente se entiende por programa un **conjunto de instrucciones ejecutables por un ordenador.**

  Un **programa estructurado** es un programa que cumple las condiciones de un algoritmo (finitud, precisión, repetición, resolución del problema,...)
- **Aplicación.** 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 de archivos forman una herramienta de trabajo en un ordenador.

Normalmente en el lenguaje cotidiano no se distingue entre aplicación y programa; en nuestro caso entenderemos que la aplicación es un software completo que cumple la función completa para la que fue diseñado, mientras que un programa es el resultado de ejecutar un cierto código entendible por el ordenador.

(1.4.2) historia del software. La crisis 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.

(1.4.3) el ciclo de vida de una aplicación

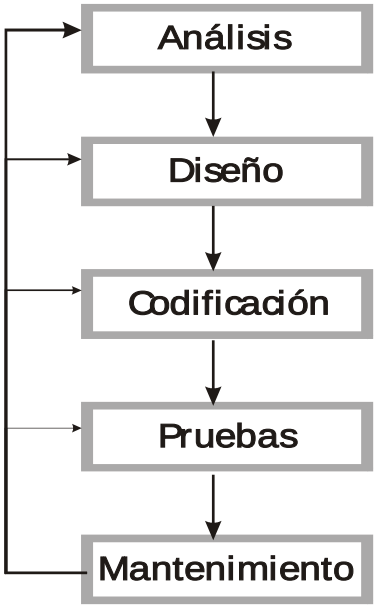


Ilustración 4, Ciclo de vida de una aplicación

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 (como ocurre con muchos de los programas distribuidos libremente por Internet).

(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.

(1.5) 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.
- **Error del programador.** Son errores que ha cometido el programador al generar el código. La mayoría de errores son de este tipo.
- **Errores de documentación**. Ocurren cuando la documentación del programa no es correcta y provoca fallos en el manejo
- **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 inesperado que el software no puede controlar (el ordenador se cuelga, errores en la grabación de datos,...)
- **Error de ejecución.** Ocurren cuando la ejecución del programa es más lenta de lo previsto.

La labor del programador es predecir, encontrar y subsanar (si es posible) o al menos controlar los errores. Una mala gestión de errores causa experiencias poco gratas al usuario de la aplicación.

(1.6) lenguajes de programación

(1.6.1) breve historia de los lenguajes de programación

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 almacenaban 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 ya 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, y el número 0001 significa restar. Una instrucción máquina que sumara el número 8 (00001000 en binario) al número 16 (00010000 en binario) sería:

```
0000 00001000 00010000
```

Realmente no habría espacios en blanco, 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.

Ejemplo[1] (programa que saca el texto "Hola mundo" por pantalla):

```
DATOS SEGMENT
   saludo db "Hola mundo!!!","$"
DATOS ENDS
CODE SEGMENT
   assume cs:code,ds:datos
START PROC
   mov ax,datos
   mov ds,ax
   mov  dx,offset saludo
   mov  ah,9
   int  21h
   mov  ax,4C00h
   int  21h
START ENDP
CODE ENDS
   END START
```

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 requiere miles y miles de 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 matemáticos. Por ejemplo la forma en FORTRAN de escribir el texto ***Hola mundo*** por pantalla es:

```
PROGRAM HOLA
 PRINT *, '¡Hola, mundo!'
END
```

[1] Ejemplo tomado de la página **http://www.victorsanchez2.net**

Poco a poco fueron evolucionando los lenguajes formando lenguajes cada vez mejores (ver). Así en 1958 se crea **LISP** como lenguaje declarativo para expresiones matemáticas.

Programa que escribe ***Hola mundo*** en lenguaje LISP:

```
(format t "¡Hola, mundo!")
```

En 1960 la conferencia **CODASYL** se creo el **COBOL** como lenguaje de gestión en 1960. En 1963 se creo **PL/I** el primer lenguaje que admitía la multitarea y la programación modular. En COBOL el programa ***Hola mundo*** sería éste (como se ve es un lenguaje más declarativo):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
MAIN SECTION.
DISPLAY "Hola mundo"
STOP RUN.
```

BASIC se creo en el año 1964 como lenguaje de programación sencillo de aprender en 1964 y ha sido, y es, uno de los lenguajes más populares. En 1968 se crea **LOGO** para enseñar a programar a los niños. **Pascal** se creo con la misma idea académica pero siendo ejemplo de lenguaje estructurado para programadores avanzados. El creador del Pascal (**Niklaus Wirdth**) creo **Modula** en 1977 siendo un lenguaje estructurado para la programación de sistemas (intentando sustituir al **C**).

Programa que escribe por pantalla ***Hola mundo*** en lenguaje Pascal):

```
PROGRAM HolaMundo;
BEGIN
  Writeln('¡Hola, mundo!');
END.
```

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 cuarto nivel. 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 éste 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++,...**

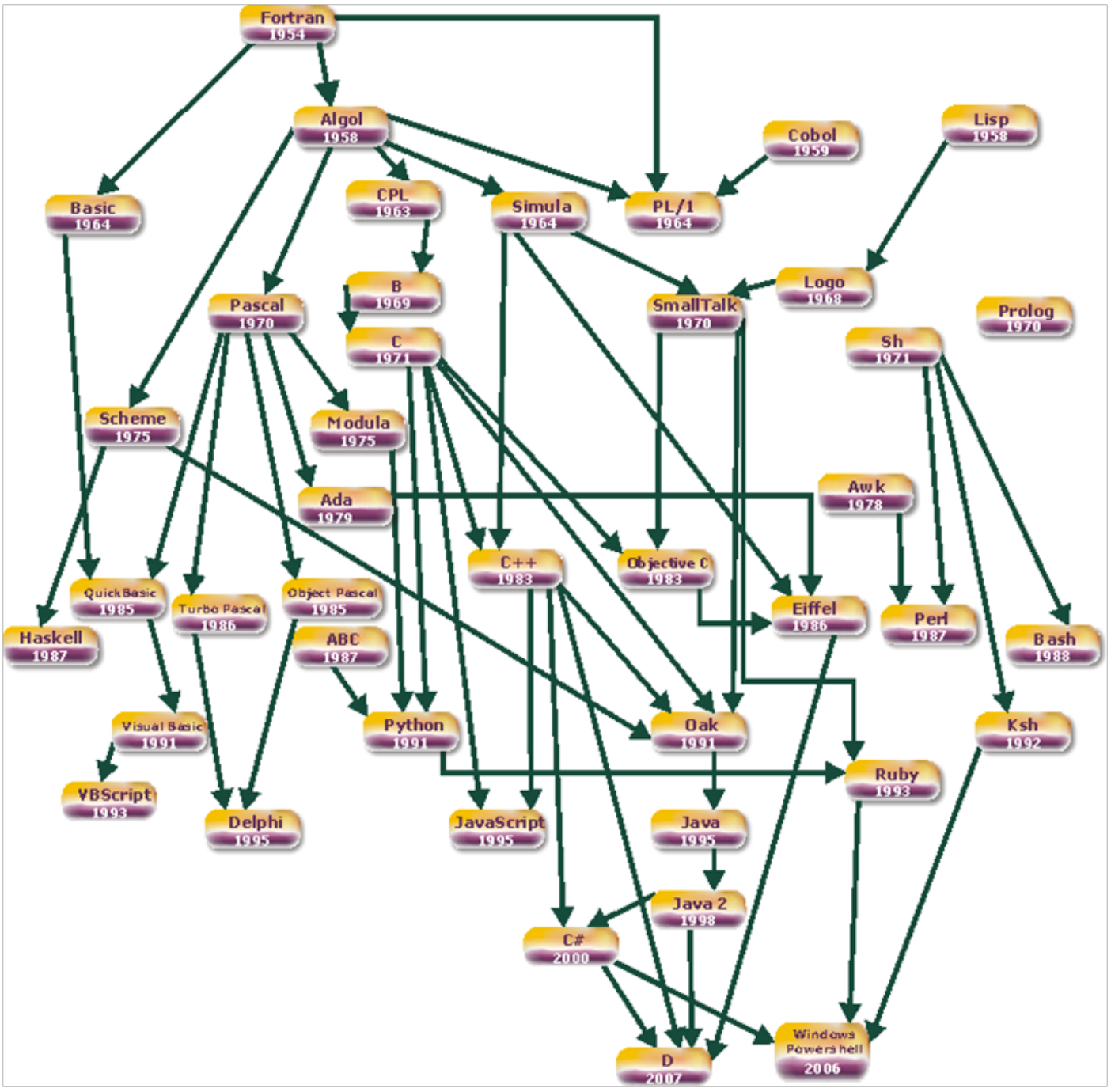


Ilustración 5, Evolución de algunos lenguajes de programación

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.

El programa ***Hola mundo*** en C# sería:

```
using System;

class MainClass
{
  public static void Main()
  {
    Console.WriteLine("¡Hola, mundo!");
  }
}
```

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. Éstos 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 página web y lenguajes de programación sencillos.

Ejemplo, página web escrita en lenguaje HTML y JavaScript que escribe en pantalla “Hola mundo” (de color rojo aparece el código en JavaScript):

```
<html>
<head>
  <title>Hola Mundo</title>
</head>
<body>
  <script type="text/javascript">
    document.write("¡Hola mundo!");
  </script>
</body>
</html>
```

(1.6.2) tipos de lenguajes

Según el estilo de programación se puede hacer esta división:

- **Lenguajes imperativos.** Son lenguajes donde 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 concentran más en el qué, que en el cómo (cómo resolver el problema es la pregunta a realizarse cuando se usan lenguajes imperativos). Los lenguajes que se programan usando la pregunta ¿qué queremos? son los declarativos. El más conocido de ellos es el lenguaje de consulta de Bases de datos, **SQL**.
- **Lenguajes funcionales.** Definen funciones, expresiones que nos responden a través de una serie de argumentos. Son lenguajes que usan expresiones matemáticas, absolutamente diferentes del lenguaje usado por las máquinas. 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**.

(1.6.3) intérpretes

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

En el caso de los 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** (o incluso, en parte, **Java**) son interpretados.

proceso

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

- 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

- 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.

(1.6.4) 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

- 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

- 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.

(1.7) programación. tipos de programación

(1.7.1) introducción

La programación consiste en pasar algoritmos a algún lenguaje de ordenador a fin de que pueda ser entendido por el ordenador. La programación de ordenadores comienza en los años 50 y su evolución a pasado por diversos pasos.

La programación se puede realizar empleando diversas **técnicas** o métodos. Esas técnicas definen los distintos tipos de programaciones.

(1.7.2) programación desordenada

Se llama así a la programación que se realizaba en los albores de la informática (aunque desgraciadamente en la actualidad muchos programadores siguen empleándola). En este estilo de programación, predomina el ***instinto*** del programador por encima del uso de cualquier método lo que provoca que la corrección y entendimiento de este tipo de programas sea casi ininteligible.

Ejemplo de uso de esta programación (listado en Basic clásico):

```
10 X=RANDOM()*100+1;
20 PRINT “escribe el número que crees que guardo”
30 INPUT N
40 IF N>X THEN PRINT “mi numero es menor” GOTO 20
50 IF N<X THEN PRINT “mi numero es mayor” GOTO 20
60 PRINT “¡Acertaste!”
```

El código anterior crea un pequeño juego que permite intentar adivinar un número del 1 al 100.

(1.7.3) programación estructurada

En esta programación se utiliza una técnica que genera programas que sólo permiten utilizar tres estructuras de control:

- **Secuencias** (instrucciones que se generan secuencialmente)
- **Alternativas** (sentencias ***if***)
- **Iterativas** (bucles condicionales)

El listado anterior en un lenguaje estructurado sería (listado en Pascal):

```
PROGRAM ADIVINANUM;
USES CRT;
VAR x,n:INTEGER;
BEGIN
   X=RANDOM()*100+1;
   REPEAT
         WRITE("Escribe el número que crees que guardo");
         READ(n);
     IF (n>x) THEN WRITE("Mi número es menor");
     IF (n>x) THEN WRITE("Mi número es mayor");
   UNTIL n=x;
WRITE("Acertaste");
```

La ventaja de esta programación está en que es más legible (aunque en este caso el código es casi más sencillo en su versión desordenada). **Todo programador debería escribir código de forma estructurada.**

(1.7.4) programación modular

Completa la programación anterior permitiendo la definición de módulos independientes cada uno de los cuales se encargará de una tarea del programa. De este forma el programador se concentra en la codificación de cada módulo haciendo más sencilla esta tarea. Al final se deben integrar los módulos para dar lugar a la aplicación final.

El código de los módulos puede ser invocado en cualquier parte del código. Realmente cada módulo se comporta como un subprograma que, partir de unas determinadas entradas obtienen unas salidas concretas. Su funcionamiento no depende del resto del programa por lo que es más fácil encontrar los errores y realizar el mantenimiento.

(1.7.5) programación orientada a objetos

Es la más novedosa, se basa en intentar que el código de los programas se parezca lo más posible a la forma de pensar de las personas. Las aplicaciones se representan en esta programación como una serie de objetos independientes que se comunican entre sí.

Cada objeto posee datos y métodos propios, por lo que los programadores se concentran en programar independientemente cada objeto y luego generar el código que inicia la comunicación entre ellos.

Es la programación que ha revolucionado las técnicas últimas de programación ya que han resultado un importante éxito gracias a la facilidad que poseen de encontrar fallos, de reutilizar el código y de documentar fácilmente el código.

Ejemplo (código Java):

```
/**
 *Calcula los primos del 1 al 1000
*/
public class primos {
   /** Función principal */
   public static void main(String args[]){
      int nPrimos=10000;
      boolean primo[]=new boolean[nPrimos+1];
      short i;
      for (i=1;i<=nPrimos;i++) primo[i]=true;
      for (i=2;i<=nPrimos;i++){
         if (primo[i]){
            for (int j=2*i;j<=nPrimos;j+=i){
               primo[j]=false;            }
         }
      }
      for (i=1;i<=nPrimos;i++) {
         System.out.print(" "+i);
      }
   }
}
```

(1.8) índice de ilustraciones