UNIVERSIDAD DE MALAGA
DPTO. DE LENGUAJES Y C. DE LA COMPUTACION
E.T.S. DE INGENIERIA INFORMATICA

FUNDAMENTOS DE LA PROGRAMACIÓN

TEMA I

INTRODUCCIÓN A LA PROGRAMACIÓN

I. 1. LA INFORMÁTICA Y EL PAPEL DE LA PROGRAMACIÓN DE COMPUTADORES.

La informática es una disciplina que nació con la aparición de los primeros computadores y cuyo crecimiento ha sido espectacular en las últimas décadas.

Es una disciplina en continua evolución tecnológica, con aplicaciones en todos los campos del quehacer humano: bionformática, informática jurídica, informática educativa,…

En los últimos años se ha producido una sinergia entre la informática, las comunicaciones y la electrónica, que ha provocado la aparición de las denominadas Tecnologías de la Información y las Comunicaciones, TIC.

La palabra Informática es de origen francés y tiene su origen en la unión de dos palabras: Información y Automática. INFORMÁTICA = INFORmación + autoMÁTICA

La definición que da la Real academia de la lengua española es la siguiente:"Conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de computadores electrónicos"

La informática es simultáneamente una disciplina matemática, científica y una ingeniería Por este motivo en ella confluyen las tres formas de pensar propias de la:

-Teoría: Usando, desarrollando y entendiendo los principios matemáticos que se usa en la disciplina informática (matemática discreta, computabilidad, gramáticas formales y autómatas,….)

-Abstracción: Esta forma de pensar tiene su origen en las ciencias experimentales y el método científico. Con la abstracción se hacen predicciones a través de los resultados teóricos obtenidos a partir de los modelos.

-Diseño: Tiene que ver con el aspecto ingenieril de la informática. Se construyen artefactos (aplicaciones, computadores, sistemas operativos, traductores, etc.….) de un modo sistemático y aplicando técnicas bien definidas

La programación es un actividad transversal asociada a cualquier área de la informática, aunque existe un área específica, la *Ingeniería del Software*, que se ocupa de toda la problemática de la creación de software. Cuando se discute la construcción del software hay que tener en cuenta que los programas pueden ser clasificados en diferentes niveles según su tamaño. Así tenemos los programas pequeños, que resuelven problemas elementales de pequeña envergadura, los programas de tamaño medio escritos por una persona durante unos cuantos meses (ejemplo proyectos fin de carrera), y los programas grandes realizados por equipos de programadores durante uno o más años. El diseño y la construcción de estos últimos entrarían dentro del ámbito de la Ingeniería del Software.

Este curso se ocupa de los primeros pasos que ha de dar una persona que quiere aprender a programar computadores y a diseñar algoritmos para resolver problemas elementales. El objetivo del mismo es proporcionar una buena base para adquirir los conocimientos que constituyen una formación en programación: estudio de los esquemas algorítmicos y estructuras de datos más importantes.

Al principio (años 50 o 60 del siglo pasado), la programación de ordenadores se veía como un arte, todo era cuestión de dominar un lenguaje de programación y aplicar las habilidades personales de resolución de problemas. En esta época se daba más importancia al hardware que estaba continuamente cambiando. Sin embargo, y según fue creciendo el número de sistemas basados en computadores y la complejidad de los mismos, los programadores empezaron a tener problemas con el mantenimiento de los programas, la corrección de errores, la adaptación a nuevos requisitos,…. El esfuerzo dedicado al mantenimiento empezó a absorber recursos de forma alarmante, y se puso de manifiesto la importancia del software

y necesidad de técnicas de análisis que facilitaran la creación del mismo.

En 1968 se reconoció oficialmente una "crisis del software" y como respuesta a ella nació la disciplina de la Ingeniería del Software. Para ello se consideró que al igual que con otros artefactos propios de la ingeniería, el software era un artefacto cuya construcción debía estar sujeta a los procedimientos rigurosos propios de la ingeniería.

Desde esta fecha y hasta la actualidad se ha avanzado mucho. Hoy se disponen de mejores herramientas (lenguajes, compiladores y entornos de desarrollo de software), de programadores mejor formados, así como de nuevas técnicas que facilitan la creación de software: la tecnología de componentes software, el desarrollo de software dirigido por modelos,…

.

I. 2. EL COMPUTADOR: UNA MÁQUINA QUE PROCESA INFORMACIÓN

Un computador es un máquina que ejecuta de forma automática una secuencia de instrucciones, que especifica qué acciones tendrá que ejecutar y en qué orden para realizar una determinada tarea.

La ejecución o procesamiento de las instrucciones, supone la transformación de unos datos de entrada en unos datos de salida o resultados. A este proceso se la denomina *Procesamiento automático de la información*.

Computador

Los datos son conjuntos de símbolos utilizados para expresar o representar un valor numérico, un hecho, un objeto, o una idea, en la forma adecuada

para su tratamiento. Esto implica que es un concepto más amplio que en matemáticas o física. Hay dos formas de representar los datos:

Datos de Entrada:

-Magnitudes físicas: datos captados directamente por el computador.

-Representación conceptual mediante caracteres.

Datos de Salida:

-Mostrados de forma física.

-Mostrados mediante caracteres.

A continuación mostramos algunos de los conceptos que constituyen la terminología básica relacionada con los computadores y la programación. A lo largo del tema desarrollaremos estos conceptos con mayor profundidad:

Software: Es el conjunto de programas ejecutables por el computador, así como todo lo relacionado con la construcción de los mismos.

Hardware: Es la parte física de la máquina, esto es el conjunto de circuitos electrónicos y parte mecánica.

Instrucción: conjunto de símbolos que representan una orden de operación para el computador. Manipulan los datos.

Programa: secuencia de instrucciones, ejecutables, directa o indirectamente, por un computador, que resuelven un problema o llevan a cabo una tarea.

Lenguaje de Programación: Establece las reglas para escribir programas.

Lenguaje Máquina: Aquel cuyas instrucciones son ejecutables directamente por el computador.

Lenguaje de Alto Nivel: Notación que permite describir los programas a un mayor nivel de abstracción que con el Lenguaje

Máquina. Es un lenguaje más cercano al programador que a la máquina. Sus instrucciones deben ser traducidas a instrucciones del Lenguaje Máquina.

Traducción entre Lenguajes: Un programa escrito en un lenguaje se transforma en el equivalente escrito en otro lenguaje.

I 3. CODIFICACIÓN DE LA INFORMACIÓN

En informática es frecuente codificar la información. La codificación es una transformación que me permite representar los elementos de un conjunto mediante los de otro, de tal forma que a cada elemento del primer conjunto le corresponda un elemento distinto del segundo. La información es necesario codificarla debido a que de esta forma es posible estructurarla y comprimirla.

En el interior de los computadores la información se almacena y se transfiere de un sitio a otro según un código que sólo usa dos valores, representados por el 0 y el 1, y que se denomina código binario. En la entrada/salida del computador se efectúan los cambios de código oportunos para que en su exterior la información sea directamente comprendida por los usuarios. La razón de usar el código binario es que en un computador los circuitos generan pulsos temporizados de naturaleza eléctrica. El computador reacciona ante la presencia o ausencia del pulso. Por este motivo el sistema binario es ideal para tales patrones, debido a que sólo usa dos valores.

Código Binario: Es un sistema digital con dos estados.

BIT (BInary digiT): unidad elemental de información. Sólo admite dos valores 0 y 1.

BYTE: 8 bits. Número de bits necesarios para almacenar un carácter.

Unidades de información (del byte)			
Sistema Internacional (decimal)		ISO/IEC 80000-13 (binario)	
Múltiplo (símbolo)	SI	Múltiplo (símbolo)	ISO/IEC
kilobyte (kB)	10^3	kibibyte (KiB)	2^{10}
megabyte (MB)	10^6	mebibyte (MiB)	2^{20}
gigabyte (GB)	10^9	**gibibyte** (GiB)	2^{30}
terabyte (TB)	10^{12}	tebibyte (TiB)	2^{40}
petabyte (PB)	10^{15}	pebibyte (PiB)	2^{50}
exabyte (EB)	10^{18}	exbibyte (EiB)	2^{60}
zettabyte (ZB)	10^{21}	zebibyte (ZiB)	2^{70}
yottabyte (YB)	10^{24}	yobibyte (YiB)	2^{80}

I.3.1. Representación posicional de los números

Un sistema de numeración en base "b" utiliza para representar los números un alfabeto compuesto por b símbolos o cifras. Cada cifra que compone un número contribuye con un valor que depende de:

a) La cifra en sí.

b) La posición dentro del número.

Ejemplo: En el sistema decimal (b=10)

{0,1,2,3,4,5,6,7,8,9}

El número 3278.52 puede obtenerse como:

$$3278.52 = 3\ 10^3 + 2\ 10^2 + 7\ 10^1 + 8\ 10^0 + 5\ 10^{-1} + 2\ 10^{-2}$$

Generalizando, dado el número... $n_4\ n_3\ n_2\ n_1\ n_0.\ n_{-1}\ n_{-2}$ ..., representado en un sistema de numeración en base b, su valor es:

$$N = ...+ n_4\ b^4 + n_3\ b^3 + n_2\ b^2 + n_1\ b^1 + n_0\ b^0 + n_{-1}\ b^{-1} + n_{-2}\ b^{-2} +...$$

(Expresión I.1)

Sistema de numeración en base 2 o binario.

Aquí b = 2 y sólo se necesitan dos símbolos para representar cualquier número: {0,1}

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

Tabla I.1

Conversión de binario a decimal.

Aplicar la expresión I.1 para b = 2.

Ejemplos:

$110100)_2 = 52)_{10}$

$10100.001)_2 = 20.125)_{10}$

En realidad basta con sumar los pesos (2^i) de las posiciones (i) en las que hay un 1.

Conversión de decimal a binario.

Aplicar el método de las "divisiones y multiplicaciones".
Ejemplo:

$26.1875)_{10} = 11010.0011)_2$

Para la parte entera:

```
26 | 2
 0  13 | 2
     1  6 | 2
        0  3 | 2
           1  1 | 2
              1  0
```

Para la parte fraccionaria:

```
0.1875  0.3750  0.7500  0.5000
   x2      x2      x2      x2
------  ------  ------  ------
0.3750  0.7500  1.5000  1.0000
```

Para la parte entera se utilizan los restos de las divisiones, en el orden inverso al que se han ido calculando.
Para la parte fraccionaria se utilizan las partes enteras de los distintos resultados de los productos, en el mismo orden en el que se han calculado.

I.3.2. Códigos de E/S

Los códigos de E/S o códigos externos asocian a cada carácter (alfabético, numérico o especial) una determinada combinación de bits. Los caracteres especiales o de control no representan datos, sino instrucciones que el computador puede ejecutar de forma automática. Ej. salto de línea, pitido,..

código E/S : a ----> b

a = {0,1,2,...,8,9,A,B,...,Y,Z,a,b,...,y,z,*,",/,...}
b = $\{0,1\}^n$

Para codificar m símbolos distintos se necesitan n bits, siendo n >= $\log_2$ m (expresión I.2)

En la práctica n es entero, y concretamente el menor número entero que verifica dicha relación.

La entrada/salida de caracteres se codifica numéricamente según una tabla de correspondencia.

Se podrían establecer códigos de E/S de forma arbitraria. Obviamente existen códigos normalizados. A lo largo de la historia se han usado diversos códigos (EBCDIC, ASCII, ISO 8859, Unicode, …)

En la actualidad los más utilizados son el código ASCII (American Standard Code for Information Interchange) y el Unicode. El primero inicialmente utilizaba 7 bits, pero posteriormente se extendió al uso de 8 bits. Unicode utiliza 8, 16 y 32 bits, englobando a ASCII e incluyendo todos los caracteres de uso común en la actualidad (alfabetos, matemáticos, musicales, iconos, …).

Código ASCII

CARACTERES DE CONTROL

0	**NUL**	(nulo)	16	**DLE**	(escape de enlace de datos)
1	**SOH**	(comienzo de cabecera)	17	**DC1**	(control de dispositivo 1)
2	**STX**	(comienzo de texto)	18	**DC2**	(control de dispositivo 2)
3	**ETX**	(fin de texto)	19	**DC3**	(control de dispositivo 3)
4	**EOT**	(fin de transmisión)	20	**DC4**	(control de dispositivo 4)
5	**ENQ**	(pregunta)	21	**NAK**	(acuse de recibo negativo)
6	**ACK**	(acuse de recibo)	22	**SYN**	(sincronización)
7	**BEL**	(campana sonora)	23	**ETB**	(fin de bloque de transmisión)
8	**BS**	(retroceso de un espacio)	24	**CAN**	(anulación)
9	**HT**	(tabulación horizontal)	25	**EM**	(fin de medio físico)
10	**LF**	(cambio de renglón)	26	**SUB**	(carácter de sustitución)
11	**VT**	(tabulación vertical)	27	**ESC**	(escape)
12	**FF**	(página siguiente)	28	**FS**	(separador de ficheros)
13	**CR**	(retroceso del carro)	29	**GS**	(separador de grupos)
14	**SO**	(fuera de código)	30	**RS**	(separador de registros)
15	**SI**	(en código)	31	**US**	(separador de unidades)

Código ASCII

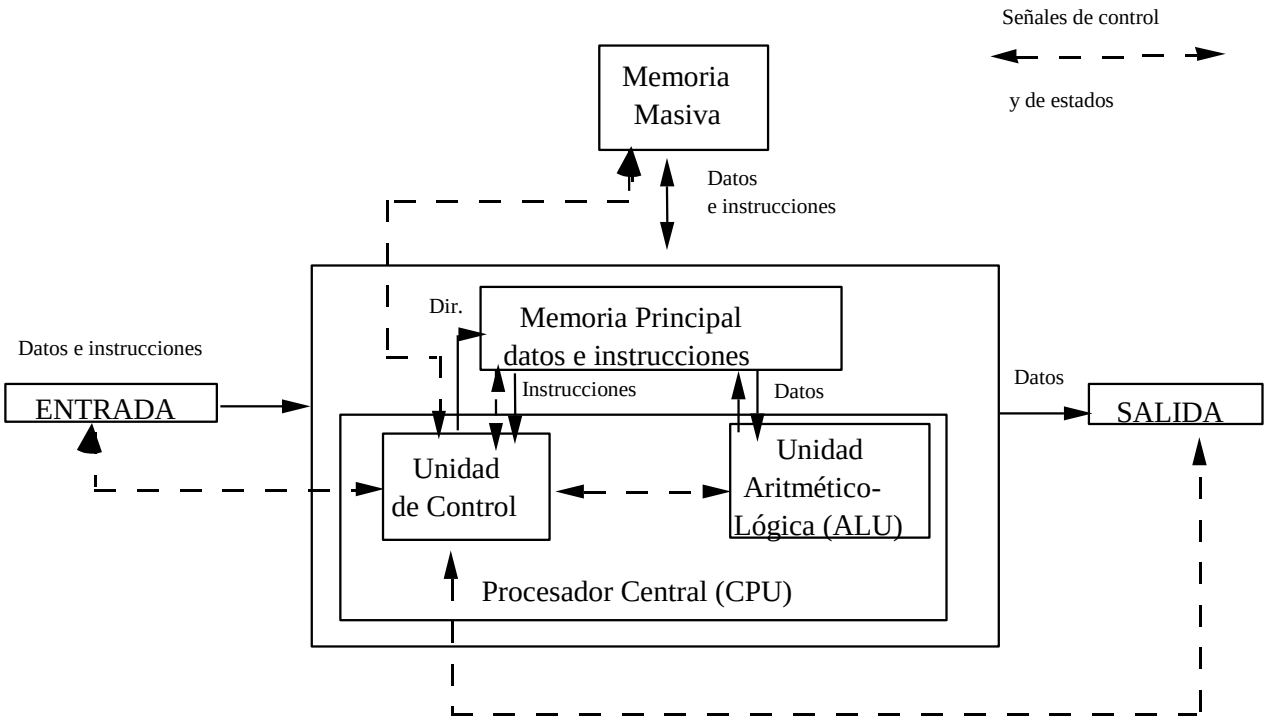
CARACTERES GRAFICOS

32	**SP**	48	**0**	64	**@**	80	**P**	96	**`**	112	**p**
33	**!**	49	**1**	65	**A**	81	**Q**	97	**a**	113	**q**
34	**"**	50	**2**	66	**B**	82	**R**	98	**b**	114	**r**
35	**#**	51	**3**	67	**C**	83	**S**	99	**c**	115	**s**
36	**$**	52	**4**	68	**D**	84	**T**	100	**d**	116	**t**
37	**%**	53	**5**	69	**E**	85	**U**	101	**e**	117	**u**
38	**&**	54	**6**	70	**F**	86	**V**	102	**f**	118	**v**
39	**'**	55	**7**	71	**G**	87	**W**	103	**g**	119	**w**
40	**(**	56	**8**	72	**H**	88	**X**	104	**h**	120	**x**
41	**)**	57	**9**	73	**I**	89	**Y**	105	**i**	121	**y**
42	*****	58	**:**	74	**J**	90	**Z**	106	**j**	122	**z**
43	**+**	59	**;**	75	**K**	91	**[**	107	**k**	123	**{**
44	**,**	60	**<**	76	**L**	92	****	108	**l**	124	**|**
45	**-**	61	**=**	77	**M**	93	**]**	109	**m**	125	**}**
46	**.**	62	**>**	78	**N**	94	**^**	110	**n**	126	**~**
47	**/**	63	**?**	79	**O**	95	**_**	111	**o**	127	**DEL**

Código ASCII Extendido

128	Ç	144	É	160	á	176	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	161	í	177	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	162	ó	178	▓	195	├	211	╙	227	π	243	≤
131	â	147	ô	163	ú	179	|	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	164	ñ	180	┤	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	165	Ñ	181	╡	198	╞	214	╓	230	µ	246	÷
134	å	150	û	166	ª	182	╢	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	167	º	183	╖	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	168	¿	184	╕	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	169	_	185	╣	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	170	¬	186	║	203	╦	219	▉	235	δ	251	√
139	ï	156	£	171	½	187	╗	204	╠	220	▄	236	∞	252	_
140	î	157	¥	172	¼	188	╝	205	═	221	▌	237	ϕ	253	²
141	ì	158	_	173	¡	189	╜	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	174	«	190	╛	207	╧	223	▀	239	∩	255	
143	Å	192	└	175	»	191	┐	208	╨	224	α	240	≡		

I 4. ESTRUCTURA FUNCIONAL DE LOS COMPUTADORES



En la figura podemos apreciar el esquema general de los primeros computadores electrónicos que seguían la arquitectura de Von Neumman. Aunque la tecnología haya evolucionado mucho, este esquema sigue siendo conceptualmente valido hoy en día. Antiguamente cada bloque correspondía a un módulo o armario independiente, conectado al resto mediante cables. Actualmente y debido principalmente al desarrollo de la microelectrónica, cada módulo se compone de unos pocos circuitos integrados o incluso uno solo, interconectados por las finas pistas de cobre de los circuitos impresos.

Características de la Arquitectura de Von Neumann:

-Datos e instrucciones se codifican en dígitos binarios.
-Tanto instrucciones como datos se almacenan juntos en la memoria principal del computador.
-El computador procesa tanto las instrucciones como los datos.

Elementos Básicos de un Computador:

Computador central (CPU): Se encarga de ejecutar las instrucciones y realizar los cálculos. Está compuesto de la Unidad de Control (UC) y la Unidad Aritmético Lógica (ALU).

a) ALU: En ésta residen los circuitos electrónicos que permiten realizar las operaciones de tipo aritmético (sumas, restas, etc.) y lógico (operaciones del álgebra de Boole, comparaciones).

b) UC: Es la que se encarga de controlar la ejecución de las instrucciones. Va recibiendo de la memoria principal las instrucciones almacenadas y va generando las señales de control necesarias para realizar las acciones a ejecutar.

Dentro de la UC tenemos el Reloj. Éste es un generador de pulsos que sincroniza todas las acciones elementales del computador. Cualquier acción del computador se realiza en un número entero de ciclos de reloj. El periodo de esta señal se denomina tiempo de ciclo. La frecuencia del reloj se mide en megahercios (MHz=Millones de ciclos/seg), y determina en parte la velocidad de funcionamiento del computador.

La UC tiene un registro denominado Registro Contador de Programa. En éste se almacena la dirección la siguiente instrucción a ejecutar.

Memoria. Permite el almacenamiento de la información e instrucciones. Sus características principales son: volatilidad, capacidad y tiempo de acceso. En función de éstas se clasifica en dos tipos:

a) Principal, central o interna**.** Tiene un gran velocidad, un alto coste y en consecuencia poca capacidad de almacenamiento. Está muy ligada a las unidades más rápidas de la computadora ALU y UC. Para que un programa se ejecute en el ordenador debe estar almacenado en memoria principal. Esta memoria se organizada en forma de vector de elementos básicos (celdas) o palabras de memoria que son identificados por su posición (dirección). Hay dos tipos:
   - RAM (Random Access Memory). Volátil, al apagarse el equipo se pierde la información). Es de lectura/escritura.
   - ROM (Read Only Memory). No volátil. Contiene información permanente. Es de sólo lectura.

b) Secundaria, externa o masiva. Esta es una memoria de poca velocidad, bajo coste y por tanto gran capacidad Es una memoria no volátil. Usa para su construcción tecnologías magnéticas y ópticas. Ej. Disco duro, pendrive,….

Palabra: La unidad de información ligada a la estructura interna del computador es la palabra. Está formada por un número entero de bytes (1,2,4 u 8) e indica la longitud de los datos con los que opera la ALU (Palabra de CPU) o los que son transferidos entre CPU y memoria (Palabra de Memoria). Ésta última coincide con el número de bits que conforma cada posición de la memoria principal, y que por tanto puede leerse o escribirse simultáneamente.

Periféricos: Son los dispositivos que el computador utiliza para comunicarse con el exterior. Hay dos tipos:

a) Entrada: Son los dispositivos por medio de los cuales se introducen al computador datos e instrucciones. Transforman la información de entrada en señales discretas o códigos. Ej. Teclado,…

b) Salida: Son los dispositivos por medio de los cuales se obtienen los resultados del computador. Transforman las señales discretas en caracteres escritos o visualizados. Ej. Pantalla, impresora,…

Buses: Son canales de información que permiten la conducción de información de un elemento a otro del computador. La información que circula a través de los buses puede ser información de control, de estado, direcciones o datos. De ahí los distintos tipos de buses. Enumeramos a continuación los más importantes:

a) Bus de datos: Es el bus por el que circulan tanto las instrucciones como los datos.
b) Bus de direcciones: Es un bus a través del cual se envían las direcciones de memoria (posiciones de la memoria donde se ha de leer o escribir) desde la CPU a la memoria. Lógicamente el ancho de este bus determina el tamaño de la memoria principal o el máximo rango de direccionamiento de la misma.
c) Bus de Control: Transfiere señales de estado y control. Gobierna el uso y el acceso a los buses de datos y direcciones. Transporta las órdenes y las señales de sincronización que vienen de la unidad de control y viajan hacia los distintos componentes de hardware.

La potencia de un computador viene determinada por: el tiempo de ciclo, la longitud de palabra y la capacidad de la memoria.

I.4.1. Funcionamiento Interno de los Computadores

Para ejecutar un programa escrito en lenguaje máquina, lo primero que hay que hacer es introducirlo en la memoria principal.

El "cargador" se encarga de introducir el programa en posiciones consecutivas de memoria a partir de una dada "i".

Una vez cargado, se le da paso a la UC, poniendo el registro CP (contador de programa) a "i", para que empiece a ejecutar el programa.

La UC repite sucesivamente las siguientes fases:

a) Fase de captación de la instrucción.

Mem(CP) --- (instrucción) ---> UC

Incremento de CP

b) Fase de ejecución de la instrucción,
y vuelta a la fase a).

Si la ejecución de una instrucción implica saltar a una instrucción distinta a la siguiente, pondrá el CP al valor de la posición de dicha instrucción, con lo que se cogerá en la siguiente fase a).

I. 5. ALGORITMOS Y RESOLUCIÓN DE PROBLEMAS

El objetivo de la Programación consiste en establecer una secuencia de acciones que tras ser ejecutadas por un procesador resuelvan un determinado problema.
Fases:

1. Análisis del problema. El primer paso requiere que el problema sea definido y comprendido, para que pueda ser analizado en profundidad. Para ello se requiere que las especificaciones de entrada y salida sean descritas con detalle.
2. Estudio de su solución.
3. Diseño del algoritmo. Para el segundo y tercer paso se utilizan técnicas que facilitan su desarrollo y se sigue una determinada metodología (diseño modular).
4. Codificación del algoritmo en un determinado lenguaje de programación. Programa. En este paso se codifica el algoritmo o la solución obtenida en un lenguaje de programación que el computador entienda y sea capaz de ejecutar.

5. Depuración y prueba. En este paso se seleccionan un conjunto de datos de prueba adecuado y se observa el comportamiento real del programa, comparándolo con su comportamiento esperado.

I.5.1. ¿Qué se quiere hacer? Concepto de Algoritmo

El término algoritmo proviene de Mohammed al-Khowârizmî, matemático persa que vivió durante el siglo IX y que alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales.
La noción de algoritmo no es particular a la informática -hay algoritmos que describen todo tipo de procesos como pueden ser una receta de cocina, una partitura musical o las instrucciones para sintonizar un televisor.
Esas operaciones sencillas se denominan *primitivas*, y deben poder ser entendidas y llevadas a cabo por un *procesador*. El procesador puede ser una persona, un computador o cualquier dispositivo electrónico o mecánico. Pero, además, el procesador debe disponer de las condiciones necesarias para poder realizar cada una de las primitivas. Esto es lo que se denomina *entorno*.

Veamos algunos ejemplos previos antes de definir el concepto de algoritmo. Consideremos los siguientes enunciados:

A) Cálculo de la media aritmética de tres números con una calculadora.

a) Pulsar la tecla C.

b) Teclear el primer número.

c) Pulsar la tecla +.

d) Teclear el segundo número.

e) Pulsar la tecla +.

f) Teclear el tercer número.

g) Pulsar la tecla /.

h) Teclear el número 3.

i) Pulsar la tecla =.

B) Preparación de un tinto de verano.

a) Colocar tres cubitos de hielo en un vaso.

b) Echar vino tinto hasta cubrir el hielo.

c) Añadir gaseosa hasta llenar el vaso.

d) Agitar el contenido.

Sin embargo, es posible que el procesador no sea capaz de interpretar una instrucción como "colocar tres cubitos de hielo en un vaso", y sea necesario descomponerla en un conjunto de acciones primitivas que éste si sea capaz de ejecutar.

a) Colocar tres cubitos de hielo en un vaso.

a1) Sacar la cubitera del congelador.

a2) Rociar la parte inferior con agua.

repetir

a3) Extraer un cubito.

a4) Echarlo al vaso.

mientras que el nº de cubitos sea menor que 3.

a5) Rellenar los huecos de la cubitera con agua.

a6) Meter de nuevo la cubitera en el congelador.

Definición de Algoritmo

- Procesador. Entidad capaz de entender una secuencia finita de acciones y ejecutar el trabajo descrito por éstas.
- Entorno. Conjunto de condiciones necesarias para la ejecución del trabajo.
- Acciones Primitivas. Son acciones que el procesador es capaz de entender y ejecutar directamente.
- Secuencialidad y Paralelismo.

Definición:

Dado un procesador bien definido y un trabajo a ejecutar por este procesador, un algoritmo es el enunciado de una secuencia finita de acciones primitivas que realizan ese trabajo.

[Kunth, 1986]. "Secuencia finita de instrucciones, reglas o pasos que describen de forma precisa las operaciones que un computador debe realizar para llevar a cabo una tarea en un tiempo finito".

Según Knuth, un algoritmo debe tener las siguientes propiedades:

- Finitud: la longitud y duración de un algoritmo son finitas.
- Precisión: un algoritmo determina sin ambigüedad las operaciones que debe ejecutar un computador.
- Efectividad: las reglas pueden ser ejecutadas por un ser humano con papel y lápiz.
- Generalidad: un algoritmo suele resolver una clase de problemas y no un problema en particular.
- Entradas y Salidas. Un algoritmo puede tener varias entradas o ninguna, pero al menos debe tener una salida que es el resultado que se pretende obtener.

A la hora de diseñar un algoritmo hay que considerar 3 aspectos:

- Primitivas de las que partimos.
- Lenguaje simbólico a utilizar.

- Representación de los datos.

En los siguientes ejemplos podemos observar cómo la representación de los datos determina la descripción del algoritmo, y por tanto la forma de las acciones.

Ejemplo: Producto de 2 números naturales X e Y.

1) Mediante un ábaco.

- Primitiva: saber contar.
- Abaco de 3 filas de bolas de distinto color.
- Representamos un dato con bolas a la izquierda.
- Inicialmente a la izquierda no hay bolas.

a) Desplazar X bolas rojas a la izquierda.

b) Desplazar Y bolas azules a la izquierda.

mientras haya bolas azules a la izquierda

c) Desplazar tantas bolas blancas a la izquierda como bolas rojas haya a la izquierda.

d) Desplazar 1 bola azul a la derecha.

fin_mientras

e) Contar las bolas blancas a la izquierda.

f) Parar

2) Mediante hojas de papel.

- Primitiva: saber sumar y restar.

- Representación de los datos mediante cifras.

- Proceso sobre 3 hojas de papel.

a) Escribir X sobre la hoja 1.

b) Escribir Y sobre la hoja 2.

c) Escribir 0 sobre la hoja 3.

mientras el valor de la hoja 2 sea mayor que 0

d) Sumar los valores de las hojas 1 y 3 y escribir el resultado en la hoja 3, perdiéndose el valor anterior.

e) Restar 1 al valor escrito en la hoja 2 y escribir el resultado en la misma hoja, perdiéndose el anterior.

fin_mientras

f) Leer el valor escrito en la hoja 3.

g) Parar.

I.5.2. ¿Qué se puede hacer? Calculabilidad y Complejidad

Calculabilidad.

Durante muchos años existió la creencia de que no existía ningún problema que no pudiera resolverse.

El matemático David Hilbert (1862-1943) quiso desarrollar un sistema matemático formal en el cual se pudiera expresar cualquier problema en términos de sentencias que fueran verdaderas o falsas. Su idea era desarrollar un algoritmo de manera que dada una sentencia en el sistema formal se determinase si dicha sentencia era verdadera o no (en definitiva un algoritmo que nos permitiera para resolver cualquier problema). Durante muchos años se intento buscar ese algoritmo sin éxito. No fue hasta 1931 cuando Kurt Gödel demostró matemáticamente que para los

números naturales ese algoritmo no puede existir (teorema de incompletitud), y que por tanto el problema de "decisión" de Hilbert no es computable.

Otros matemáticos como Church, Kleene, Post y Turing encontraron otros problemas no computables, es decir, problemas para los no se puede encontrar un algoritmo que los resuelva. Estos resultados fueron expuestos a principios de los años 30 cuando aún no se había desarrollado ningún computador.

Por tanto, existen problemas no computables, esto es, no existe y nunca existirá ningún algoritmo que los resuelva. Algunos ejemplos de problemas no computables:

-El problema de la Parada (the halting problem).
-El problema del Castor Afanoso (busy beaver problem).

El problema de la parada es un problema aparentemente sencillo que no es computable. En programación ocurre que cuando se está escribiendo un programa, se cometen errores que provocan que el programa nunca acabe. Veamos un ejemplo de algoritmo que nunca termina.

a) Leer un valor de teclado e introducirlo en la variable x.
repetir
  b) Restar 1 al valor de x.
mientras que el valor de x sea distinto de 0.

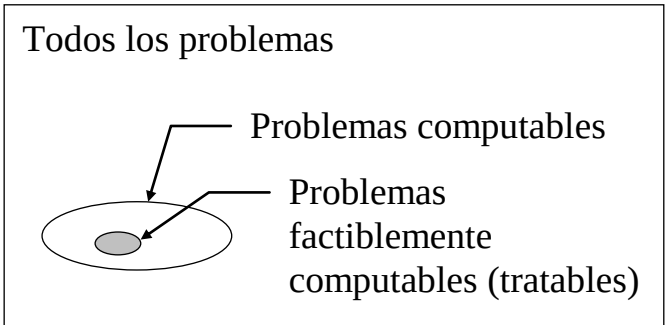
Este algoritmo que puede parecer correcto, no terminará nunca si el valor introducido por teclado es un número negativo. Sería muy útil tener un programa que nos dijera si un programa P ante unos datos de entrada D se detendrá o no. Sin embargo, no merece la pena intentar desarrollar ese programa puesto que el problema de la detención no es computable.

La *teoría de la calculabilidad* estudia y caracteriza qué problemas son algorítmicamente resolubles o computables.

Complejidad

Es interesante conocer los recursos computacionales necesarios para ejecutar un determinado algoritmo.

La *teoría de la complejidad* se encarga de estudiar la cantidad de recursos computacionales (espacio y tiempo) que necesita un determinado algoritmo para ejecutarse. Sólo aquellos algoritmos que utilizan una cantidad factible de recursos son útiles en la práctica.



Dentro de los problemas computables se distinguen dos categorías:

- Tratable: cuando existe un algoritmo que lo resuelve de manera eficiente.
- Intratable: cuando no existe un algoritmo que lo resuelva de manera eficiente.

Un ejemplo de problema computable pero intratable, es el problema del viajante de comercio. Éste consiste en encontrar el camino de menor longitud que pase una sola vez por un determinado conjunto de ciudades y vuelva al punto de partida.

Para este problema existen n! combinaciones de recorridos, y no se conoce ningún algoritmo que proporcione la de menor longitud de manera eficiente, es decir, sin tener que examinar la mayor parte de las n! soluciones. Por ejemplo, para un tamaño de n=100 ciudades hay alrededor de $3*7*10^{156}$ soluciones (incluso si todos los átomos del universo pudieran

calcular mil millones de soluciones por segundo, sería necesario mucho más tiempo del que lleva el universo existiendo).

La eficiencia de un algoritmo (parámetros espacio/tiempo) es dependiente del tamaño de los datos de entrada y también de la arquitectura del computador. Sin embargo, buscamos variaciones más abstractas del espacio y tiempo computacional que sean lo más independientes posibles del tipo de computador en el que se ejecute el algoritmo, y que nos permitan comparar la eficiencia de diferentes algoritmos que resuelvan el mismo problema.

Existen diversas medidas de la eficiencia de un programa:

- Número de instrucciones en código máquina: depende del computador y de los detalles de implementación.
- Número de instrucciones en un lenguaje de alto nivel: depende del lenguaje y del estilo de programación.
- Número de operaciones básicas: es una medida independiente.

Nos quedaremos con esta última medida de la eficiencia, y para medir la complejidad computacional de un algoritmo identificaremos aquella operación básica dentro del algoritmo que capture el grueso de la computación, y contaremos el número de veces que ésta se repite.

Ejemplo: en un algoritmo de búsqueda de un elemento dentro de una lista la operación básica (más costosa) es la comparación de elementos. Supuesto que la lista es de tamaño N obtendremos una función T(N) que expresará el tiempo de ejecución de ese algoritmo, y donde N es el tamaño de los datos sobre los que se ejecuta el algoritmo.

Si la función T(N) tiene un término N^c como factor dominante, donde c es una constante tal que c>0, se dice que es un algoritmo polinomial y tratable. Si la función tiene un término a^N como factor dominante, donde a es una constante tal que a>1, se dice que el algoritmo es exponencial e intratable, ya que sólo finaliza en un tiempo razonable para valores pequeños de N. En la siguiente tabla se muestran valores de algunas funciones para distintos tamaños de entrada. El objetivo es observar el comportamiento de dichas funciones cuando se dobla el tamaño de la entrada.

	N=10	N=20	N=40	N=80	N=160	N=320	N=640	N=1280
logN	1	1.30	1.60	1.90	2.20	2.50	2.80	3.10
N	10	20	40	80	160	320	640	1280
N^2	100	400	1600	6400	25600	102400	409600	1638400
N^4	10000	160000	$2.56x10^6$	$4.096x10^7$	$6.5536x10^8$	$1.048x10^{10}$	$1.677x10^{11}$	$2.68x10^{12}$
2^N	1024	$1.048x10^6$	$1.0995x10^{12}$	$1.20x10^{24}$	$1.461x10^{48}$	$2.1359x10^{96}$	$4.56x10^{192}$	∞

Supuesto que un computador pueda ejecutar mil millones de operaciones por segundo, si se examinan los resultados obtenidos en la tabla anterior, se puede comprobar que para un valor pequeño de N (80) el algoritmo de coste 2^N es claramente no factible.

I.5.3. ¿Cómo hay que hacerlo? Corrección

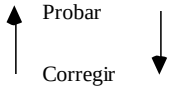
Este es un objetivo básico exigible a los programas. En la actualidad la mayoría de los programas contienen errores. Por ejemplo, la explosión del cohete espacial europeo Ariane5, fue debida a un error del programa que provocó que una operación aritmética produjese un valor inesperado. A continuación enumeramos los 3 tipos de errores que pueden estar presentes en un programa:

1) Errores lexicográficos, sintácticos y semánticos (ver sección I.6.2). Son fáciles de corregir.
2) Errores lógicos en la especificación y diseño del algoritmo: consisten en partir de una especificación incorrecta del problema que se desea resolver. Estos errores son más costosos de corregir cuanto más tarde se detecten, puesto que se desperdicia tiempo y recursos al tener que repetir el desarrollo (ej. hacemos un producto cuando queremos hacer unas suma).
3) Errores en tiempo de ejecución: errores lógicos tan severos que el sistema normalmente parará la ejecución del programa (ej. división por cero).

¿Cómo asegurar que un programa es correcto? Hay diferentes enfoques:

1) **Prueba Experimental:**

Consiste en seleccionar un conjunto adecuado de datos de entrada de prueba y observar el comportamiento real del programa, comparándolo con el comportamiento esperado del mismo. Éste es un proceso iterativo:



El problema consiste en determinar el conjunto de casos de prueba adecuado, junto con la interpretación de los mismos. Además, ¿Cómo decidir si los resultados de una prueba son correctos o no?

La prueba sólo puede concluir que un programa es correcto si se testean con éxito todas las posibles entradas al mismo. Por ello, no es factible en la práctica.

Ejemplo: Un método para calcular N^2 para algún entero positivo N sin realizar el producto consiste en sumar los N primeros impares:

$$N^2 = 1+3+5+...+(2N-1)$$

Ante cualquier algoritmo que construyamos, nos encontramos con que no es factible probar para todo N. ¿Cómo sabemos que la fórmula no fallará para algún valor no probado?

El Teorema de Gilb dice:"La prueba experimental puede ser empleada para mostrar la presencia de errores, pero nunca su ausencia". Por tanto la solución pasaría por emplear un razonamiento matemático:

$$\sum_{i=1}^{N}(2i-1)$$

$$= 2\sum_{i=1}^{N} i - N = 2\frac{N(N+1)}{2} - N = N^2 + N - N$$
$$= N^2$$

La única forma de demostrar la corrección de un programa es mediante la *verificación de programas*. La solución anterior se enmarca dentro de las técnicas de verificación formal

2) **Verificación formal:**

Consiste en demostrar mediante algún método la corrección de un programa. Éste es un enfoque análogo a la demostración de teoremas.

La corrección es una noción relativa. Cuando decimos que un programa es correcto nos referimos respecto a alguna especificación. Hay dos tipos de corrección:

- Corrección parcial. Se supone que el programa termina.
- Corrección total. Se demuestra que el programa termina.

Técnica para verificar programas. Con los asertos (expresiones lógicas) expresamos las condiciones que se le imponen a los datos. Dependiendo del punto del algoritmo en que aparezcan serán:

- *Precondiciones*. Son asertos que deben ser verdad antes de ejecutar un algoritmo o parte del mismo.
- *Postcondiciones*. Son asertos que expresan las propiedades de los resultados de los algoritmos.
- *Asertos intermedios* Son asertos situados en distintos puntos del algoritmo.

* Notación: {P} S {Q} (Especificación pre/post)

S -> Algoritmo
{P} -> Precondición de S
{Q} -> Postcondición de S

Interpretación: Si P es verdad y se ejecuta S, entonces (suponiendo que S termina) Q es verdad.

Verificar un programa consiste en demostrar que cumple su especificación pre/post.

I. 6. LENGUAJES DE PROGRAMACIÓN

Como dijimos en la introducción un lenguaje de programación establece las reglas para describir programas. La clasificación de los lenguajes de programación se puede hacer atendiendo a diferentes criterios:

1) Nivel de abstracción
2) Propósito del lenguaje
3) Paradigma de programación

1) Clasificación según el nivel de abstracción:
   - Lenguajes de bajo nivel:
     - Cercanos a la máquina.
     - Bajo nivel de abstracción. Programas expresados en términos de la arquitectura y operaciones básicas que realiza un computador (ej. Lenguaje máquina, Ensamblador,…..)
   - Lenguajes de alto nivel:
     - Cercanos al problema
     - Alto nivel de abstracción. Programas expresados en términos del problema a resolver, en un lenguaje cercano al programador (ej. Lisp, Prolog, Java, C++, Ada,…)

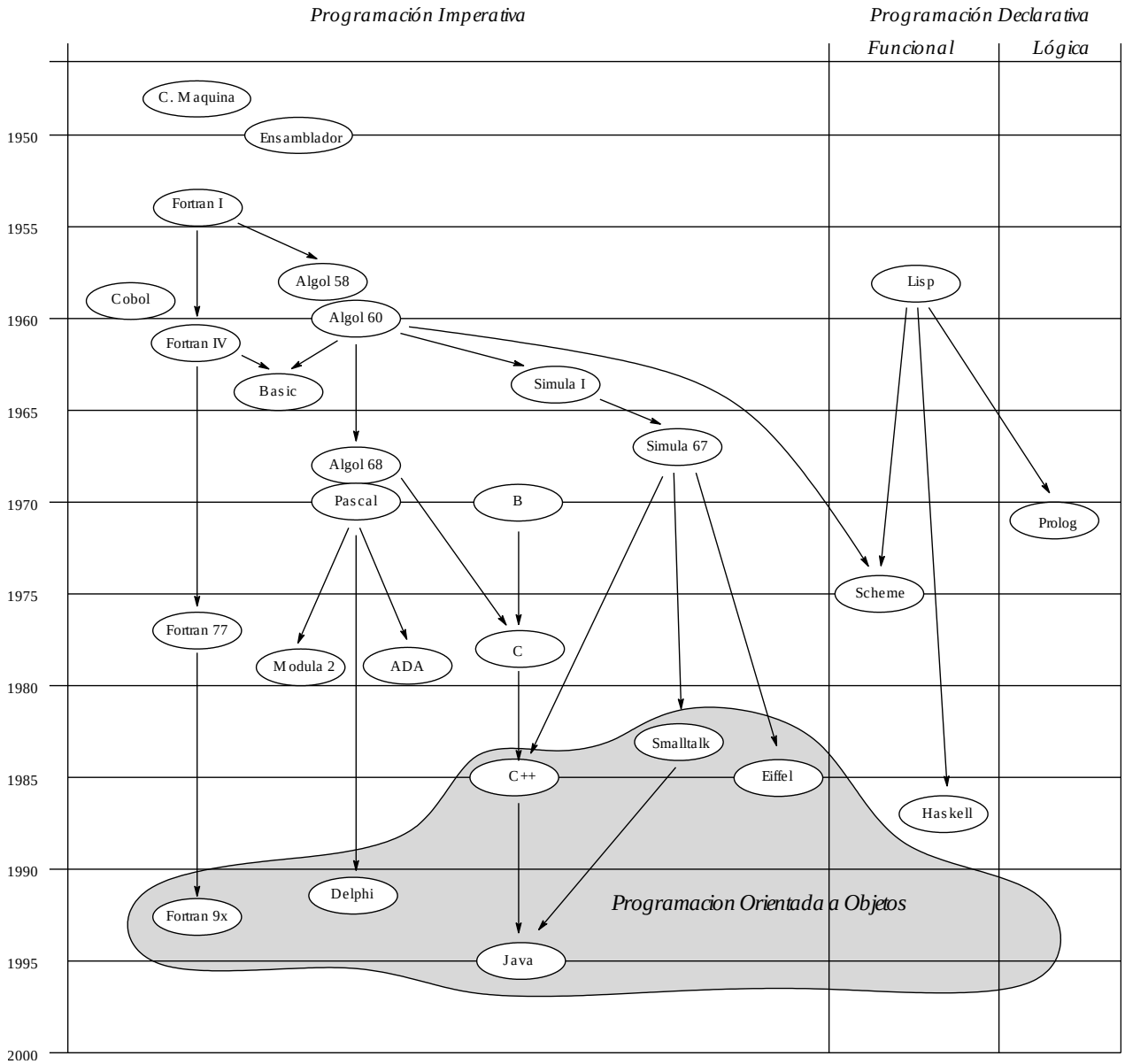
2) Clasificación según el propósito del lenguaje:
   - Científicos: para realizar cálculos matemáticos, manipulación de fórmulas,…Ej. Algol, Fortran
   - Ingenieriles: diseño de hardware, control de sistemas, simulaciones,..Ej. Ada, Dynamo, Simscript,..
   - Gestión: contabilidad, bases de datos,…Ej. Cobol, Dbase,..

- Inteligencia Artificial: sistemas expertos, planificación,..Ej. Lisp, Planner,..
- Multipropósito: aplicaciones de propósito general. Ej. Pascal, Modula-2, C,..

3) Clasificación según el paradigma de programación. Un paradigma de programación es un modelo que engloba a ciertos lenguajes que comparten:

- Elementos estructurales: ¿con qué se confeccionan los programas?
- Elementos metodológicos: ¿cómo se confecciona un programa?

	E. Estructurales	E. Metodológicos
Imperativo	Subprogramas	Refinamientos sucesivos
Orientado a Objeto	Objetos	Orientado a Objetos
Declarativo: Lógico y Funcional	Funciones (Funcional) Hechos y reglas (Lógica)	Análisis de Relaciones



En la figura anterior podemos ver la genealogía de diversos lenguajes. En los orígenes de la programación es cuando se produce el mayor caos. Posteriormente se produce una convergencia gradual hacia ciertos modelos que se imponen (imperativo, orientado a objetos,…). Como ocurre en la naturaleza hay una selección natural. Los lenguajes menos adecuados tienden a desparecer, y son sustituidos por otros lenguajes más aptos.

I.6.1. Reconocimiento de Lenguajes. Gramáticas

Asociada a cada lenguaje de programación, y al igual que en los lenguajes naturales, existe una gramática que establece la forma en la que se pueden

construir y reconocer sentencias de un programa (frases) correctas o pertenecientes a dicho lenguaje. El objetivo es que la sintaxis o notación usada para describir los lenguajes de programación no se ambigua.

Una gramática se define como una tupla $G = (N, T, A, R)$, donde:

- N: símbolos no terminales o categorías sintácticas del lenguaje, definibles en función de otros más simples.
- T: símbolos terminales o palabras del lenguaje. No se pueden definir en términos de otros más elementales.
- A: Axioma, que es un símbolo no terminal.
- R: Reglas de derivación que indican cómo se combinan dichos símbolos anteriores para construir frases o palabras correctas de acuerdo al lenguaje generado por la gramática.

Entre las diversas formas de expresar la gramática de un lenguaje, las más utilizadas son la notación BNF y los diagramas sintácticos de CONWAY.

Ejemplo: Identificadores en un determinado lenguaje. Un identificador está formado por una secuencia de cualquier longitud de letras minúsculas o números, pero ha de comenzar obligatoriamente por una letra minúscula. Ejemplo de identificadores válidos: hola, a123, p5hy. Sin embargo, 457, o 4tyu no serían identificadores válidos.
La siguiente gramática nos permite distinguir entre los identificadores válidos o que pertenecen al lenguaje generado por la gramática, y los no válidos o no perteneciente al lenguaje. La expresaremos primero en notación BNF y después en notación Conway:

a) Notación BNF:

- $G = (N, T, A, R)$
  - N = {<ident>, <letra>, <digito>}
  - T = {a,b,c,d,…,z,0,1,…9}
  - A = <ident>
  - R= 1.<letra> ::= a | b | c | . . . | z
    2. <dígito> ::= 0 | 1 | 2 | 3 | . . . | 9
    3. <ident> ::= <letra>{<letra>|<dígito>}

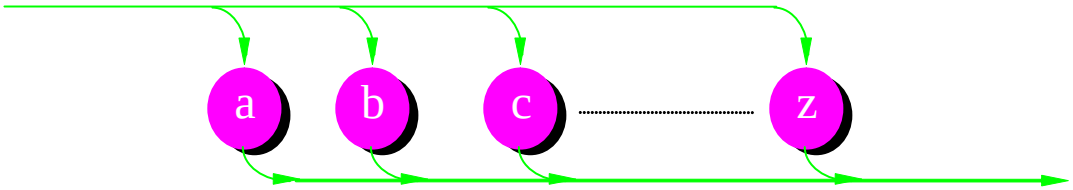
b) Notación Conway: En esta notación las Reglas, R, se representan de forma gráfica.

- Símbolo terminal 

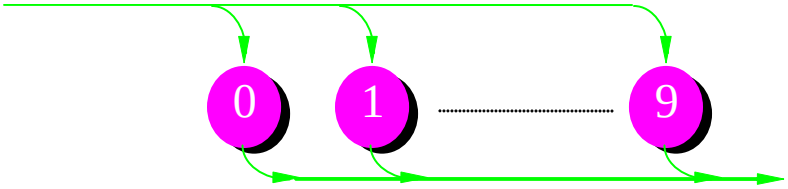
- Símbolo no terminal 

- Regla: conjunto de símbolos unidos por flechas. Cada camino de izquierda. a derecha es una derivación válida dentro del lenguaje

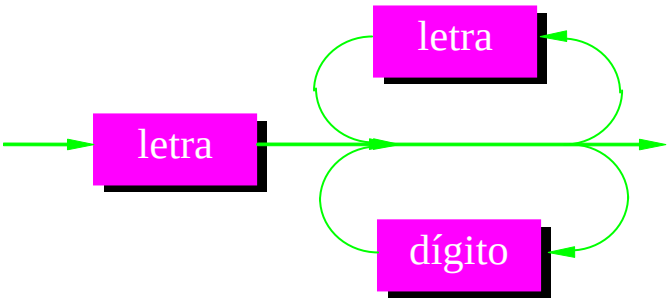
<letra>



<digito>



<ident>



I.6.2. Traductores, compiladores e intérpretes

Traductor: programa que convierte un programa escrito en un determinado lenguaje A a su equivalente escrito en otro lenguaje B.

Compilador: traductor para convertir un programa escrito en un lenguaje de alto nivel a su equivalente escrito en lenguaje máquina. Traduce el programa globalmente y genera un programa objeto como resultado de la traducción. El resultado puede ser lenguaje máquina de un computador real (Ej. compilador de C++) o bien de una máquina virtual (Ej. compilador de Java).

Intérprete: programa para conseguir que un programa escrito en lenguaje de alto nivel sea ejecutado "directamente" en la máquina. En realidad irá traduciendo y ejecutando cada instrucción del programa, sin generar programa objeto alguno.

Esquema general de un compilador

El funcionamiento de un compilador se divide en dos fases:

- Análisis: se comprueba el léxico, la sintaxis y la semántica del programa fuente.
- Síntesis: se genera un código intermedio y se optimiza, generando así el código definitivo en lenguaje máquina.

Fase de Análisis: En esta fase se hace uso de:

- Rutina de errores: que manejará los errores detectados durante la fase de análisis, mostrándolas al final al programador por pantalla.
- Tabla de símbolos: En esta tabla es donde se almacenan los objetos usados en el programa: variables, constantes,.. De cada uno de ellos se almacena su tipo, nombre, valor o posición que ocupa en memoria.

Además se divide en tres partes:

1) Análisis Lexicográfico:
   -Genera una secuencia de símbolos denominados "tokens" (unidades léxicas) que será la entrada de la siguiente subfase.
   - Elimina comentarios.
   - Elimina espacios en blanco, separadores, tabuladores,...
   - Avisa de los errores lexicográficos que detecte.

2) Análisis Sintáctico:

- Analiza la estructura de la frase (ya formada por "tokens") y comprueba si pertenece al lenguaje. Para ello crea un árbol sintáctico a partir de la secuencia de tokens de la fase anterior
- El árbol sintáctico sirve como base para el posterior análisis semántico y la generación de código
- Avisa de los errores sintácticos que detecte

3) Análisis Semántico

- Comprueba que una frase sintácticamente correcta lo es también semánticamente.
- Ejemplo: asignación de valores de distintos tipos, o aplicación de operadores a tipos apropiados
- Avisa de los errores semánticos que detecte

Fase de Síntesis: la fase se síntesis se divide en tres partes:

1) Generación de código intermedio:

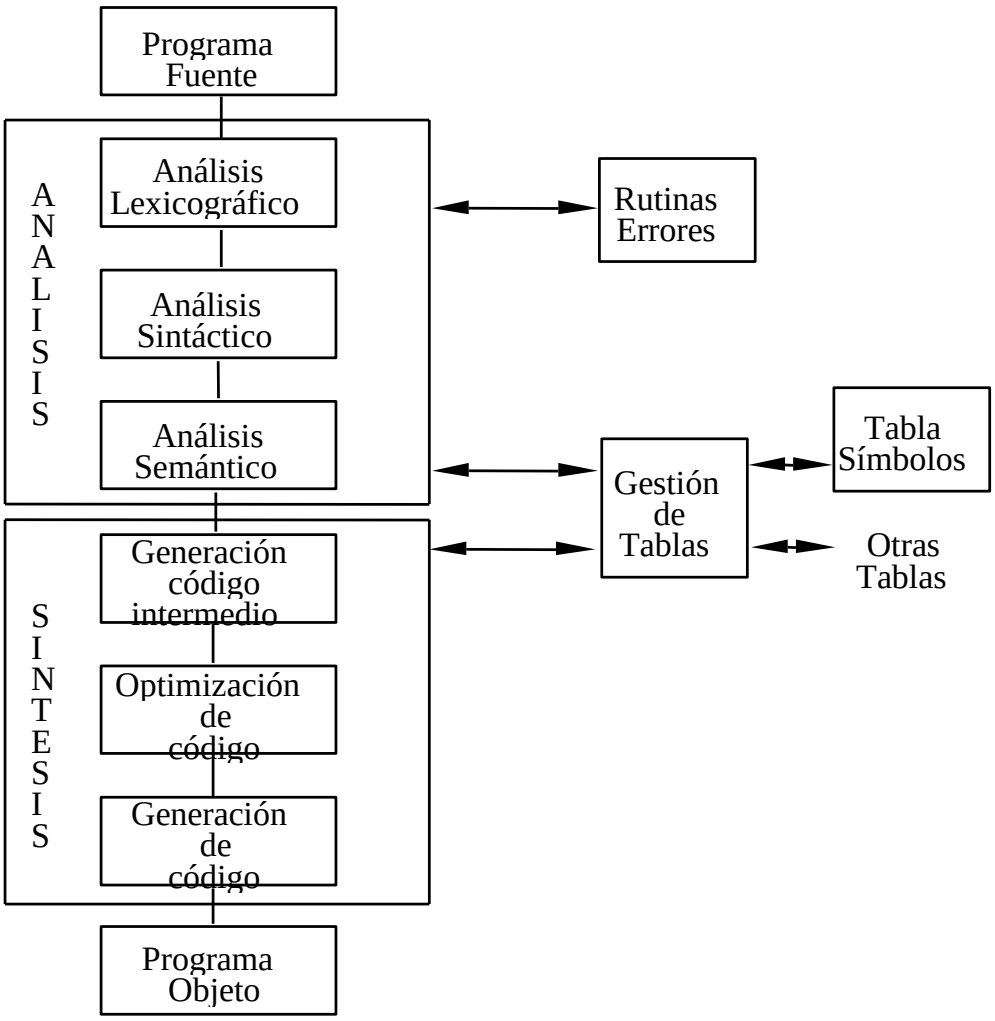
- Código independiente de la máquina para la que se hace el compilador.
- Debe ser fácil de producir a partir del análisis y fácil de traducir al código definitivo

e) Optimización de código:

- Genera un código más compacto y eficiente.

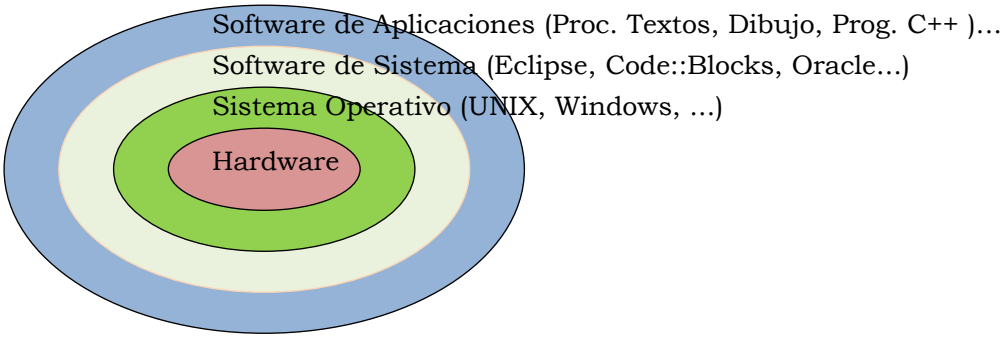
f) Generación de código:

- Pasa del código intermedio (posiblemente optimizado) al código máquina final.



I. 7. VISIÓN GENERAL DE UN SISTEMA INFORMÁTICO

En la siguiente figura se muestra un esquema simplificado de un sistema informático mediante una estructura en capas o niveles:



Ya que en la sección I.4 describimos la estructura funcional de los computadores, vamos a describir a continuación la estructura del software:

- Software de aplicación:
  - Programas de usuario: son programas destinados a resolver problemas específicos reales y que normalmente son realizados por el propio usuario del computador.
  - Programas verticales: son programas que logran resolver problemas similares a un conjunto de usuarios que pertenecen al mismo colectivo profesional o sector. Ej. programas de contabilidad general, programas de arquitectura (autocad), …
  - Paquetes estándar: programas que pueden ser útiles a numerosos colectivos y resuelven problemas que se pueden tratar de forma común. Ej. procesadores de texto, programas de diseño gráfico, ...
- Software de Sistema: conjunto de programas que proporcionan servicios para la utilización o desarrollo de programas, pero que en sí no resuelven el problema final del usuario.
  - Lenguajes, entornos de programación y traductores (IDE, Entorno integrado de desarrollo).
  - Soporte y sistema de gestión de datos (DBMS). Conjunto de programas que permiten organizar grandes volúmenes de información, y facilitan su manipulación y acceso.
  - Entornos gráficos de usuario (GUI). Conjunto de programas que permiten la comunicación con el usuario a través de elementos gráficos y textuales.
- Sistema Operativo (OS): Conjunto de programas que controlan y facilitan el uso del Hardware.

I.7.1. Entorno integrado de desarrollo

Un entorno de desarrollo integrado o IDE (acrónimo en inglés de **I**ntegrated **D**evelopment **E**nvironment), es un programa informático compuesto por un conjunto de herramientas de programación.

Componentes:

- Un editor de texto.
- Un compilador.
- Un depurador.
- ...

Puede dedicarse en exclusiva a un sólo lenguaje de programación o utilizarse para varios. En nuestras prácticas de laboratorio utilizaremos Code::Blocks, un IDE de software libre para C, C++ y Fortran.