ALGORITMOS, OBJETOS y ESTRUCTURAS DE DATOS

Una introducción a la programación con contratos.

Gerardo Rossel

Andrea V. Manna

A mis padres y a mis hijos Lara y Darío por darme todo lo necesario.

Gerardo Rossel.

A mis hijos y a mis padres, siempre allí

Andrea Manna

Tabla de Contenidos

Conceptos Introductorios

Un enfoque orientado a objetos

Este libro es adecuado para un primer o segundo curso de algoritmos y estructura de datos dependiendo del enfoque orientado a objetos que tenga la curricula. También es adecuado para un curso de programación orientada a objetos en aquellas instituciones en las cuales no se siga un enfoque "objetos primero".

¿A qué nos referimos con objetos primero? La forma de introducir a la programación a los alumnos tiene distintos enfoques. Algunos plantean la enseñanza de lenguajes funcionales porque se argumenta que los alumnos a nivel terciario y/o universitario ya cuentan con el concepto matemático de función. Otros plantean el camino más tradicional de comenzar con un lenguaje imperativo para incorporar finalmente los conceptos de clases y objetos. El enfoque "objetos primero" se basa en que la mayor parte de los desarrollos de software hoy en día se realizan sobre un lenguaje orientado a objetos (con distintos grados de rigurosidad). Y sólo algunas aplicaciones muy específicas no utilizan esta forma de desarrollar. Dado que en el mundo profesional se encontrarán programando en Java, C#, C++, etc., y que todos esos lenguajes soportan los conceptos de objetos y clases, no se ve motivo alguno para demorar el ingreso de los alumnos a la orientación a objetos. Se argumenta, además, que es más complicado el cambio de paradigma una vez que se dominaron los conceptos básicos desde un paradigma imperativo puro a orientación objetos, que al revés.

En definitiva es un tema polémico y no pretendemos dar la palabra definitiva. En el libro nos dedicamos a exponer los conceptos canónicos de un curso de programación desde un punto de vista orientado a objetos.

Los ejemplos y muchos de los conceptos son dados en dos lenguajes comerciales reales: Eiffel y C#. La elección de estos dos lenguajes se debe a la fortaleza conceptual y la difusión. Eiffel es un lenguaje potente, usado comercialmente y con una serie de características que

lo hacen ideal para aprender a programar bien. C# por otro lado, si bien no soporta todos los mecanismos de Eiffel, es un lenguaje moderno cuidadosamente diseñado y de uso masivo en el mundo profesional.

Otra de las cosas en la cual haremos énfasis durante todo el texto, es en los contratos de software. Los contratos de software son una herramienta importante en la construcción de software correcto. Eiffel cuenta con soporte incorporado de contratos de software, el cual es otro de los motivos de su elección como lenguaje ejemplo. Comenzaremos definiendo el concepto de algoritmo.

¿Qué es un algoritmo?

La algorítmica es la rama de las ciencias de la computación que se encarga del estudio de los algoritmos, su implementación, diseño, corrección y eficiencia.

***Def**. Un algoritmo es una descripción de una secuencia ordenada de acciones primitivas que pueden ser ejecutadas por una máquina (una computadora por ejemplo) y que pueda procesar ciertos datos de entrada entregando otros de salida sin otras instrucciones*

El algoritmo es, según la definición, una secuencia ordenada de pasos, es decir, el orden en el cual se realizan las acciones es importante y debe estar explícito en el lenguaje de representación del algoritmo. Esto es así en el paradigma de programación denominado imperativo. Otros paradigmas, como el de programación lógica, utilizan otras formas de representar las soluciones, incluso de manera declarativa indicando *que* se debe resolver en lugar de *cómo*.

A su vez, un algoritmo, debe contar con una representación adecuada de los datos que necesite manipular. La elección adecuada de esta representación de los datos, también llamada *estructura de datos*, es crítica para el diseño de un algoritmo correcto y de calidad.

Este libro tratará fundamentalmente sobre las técnicas para implementar algoritmos computacionales sobre estructuras de datos.

La técnica que permite, a partir de un algoritmo (solución a un problema definido) transformarse en un programa plausible de ser llevado a una computadora, es denominada *refinamiento sucesivo*. Para explicar el concepto de refinamiento sucesivo, es necesario primero definir el concepto de *sentencia primitiva*. Una sentencia (que indica una acción a ser llevada a cabo por un procesador) es primitiva si el procesador que debe resolver el problema puede ejecutarla sin mayor información. Una sentencia *no primitiva* pude descomponerse en varias sentencias primitivas. La técnica de llevar un algoritmo especificado como un conjunto de sentencias no primitivas a un conjunto de sentencias primitivas es la que denominamos *refinamiento sucesivo o progresivo*.

Definir una sentencia como primitiva es dependiente de la máquina que finalmente ejecute el algoritmo o mejor dicho, del lenguaje de programación en el cual se implemente. Distintos lenguajes de programación proveen diversos niveles de abstracción. Lo que en el lenguaje ensamblador (que tiene una correspondencia uno a uno con las instrucciones en código máquina, es decir aquellas que el procesador puede entender) requiere de varias sentencias, en un lenguaje de alto nivel como el Pascal puede ser resuelto en una sola línea de código.

Nuestros algoritmos van a ir acompañados de la estructura de datos que manipulan lo cual nos lleva a un enfoque basado en el paradigma orientado a objetos de la programación. Un paradigma de programación es un conjunto de esquemas conceptuales que definen una manera de razonar, implementar algoritmos y construir programas.

El paradigma de orientación a objetos se basa a su vez en el paradigma imperativo. Este último tiene su fundamento en el modelo de

computador conocido como de *Von Neumann*. El computador de *Von Neumann* se compone de un procesador, una memoria y elementos auxiliares. Este es un modelo teórico general que utilizan muchos de los computadores físicos reales y basado en él, se diseñaron la mayoría de los lenguajes de alto nivel.

El modelo tiene un funcionamiento cíclico en el cual el procesador realiza una acción después de otra en forma secuencial, cada acción puede eventualmente alterar el estado de la memoria. El procesador es el encargado de realizar las acciones y cuenta con un número limitado de operaciones posibles.

Tanto las operaciones como los datos necesarios se encuentran almacenados en la memoria del computador, que se estructura como celdas consecutivas y numeradas (direcciones). La memoria usa el formato binario para guardar la información. El orden de ejecución se determina por la posición en la memoria (salvo indicación en contrario).

El pseudocódigo utilizado para representar los algoritmos imperativos, omite detalles del procesador y la implementación concreta. Los datos se almacenan en *variables* o *constantes* y las operaciones posibles se denominan *instrucciones*. La instrucción principal es la *asignación*, la cual asigna un valor a una variable. Por ejemplo en el lenguaje Eiffel la forma de asignar es la siguiente:

```
variable := valor
```

Un ejemplo:

```
v := 4
```

En este ejemplo, a una variable de nombre `v` le estamos asignando el valor `4`. Luego veremos los valores posibles y sus significados. El símbolo `:=` es denominado símbolo de asignación, a la izquierda se encuentra la variable y a la derecha el valor.

El importante notar que el significado de la instrucción depende del lugar donde se encuentre la variable. Por ejemplo:

```
v := v + 1
```

En este ejemplo la variable `v` aparece dos veces, a la izquierda y a la derecha. La lectura de la sentencia es la siguiente: al valor actual de la variable `v` se le suma el valor `1` y luego el resultado es almacenado en la variable `v`. Es decir, primero se calcula la parte derecha y luego se realiza la asignación. En esta sentencia vemos otra operación: la suma. Naturalmente que para que esto sea correcto, la variable `v` debe tener almacenado un número. El hecho de que la variable tenga almacenado un número puede ser verificado de antemano si declaramos a la variable `v` como que solamente puede almacenar números. Esto se denomina tipificar, es decir dar un *tipo*. Los tipos y variables forman lo que se denomina ambiente. Cada variable puede ser accedida dentro del ambiente en el cual fue declarada y según los límites que imponga el lenguaje de programación

Compilación de Programas

Si bien existen diversos lenguajes de alto nivel, los computadores sólo pueden entender el código máquina, es decir, un conjunto de instrucciones sumamente primitivas diseñadas para ese procesador.

Básicamente, tenemos tres tipos de lenguajes:

- Lenguajes de máquina
- Lenguajes ensambladores
- Lenguajes de alto nivel.

En un *lenguaje de máquina*, cada instrucción es una secuencia de ceros y unos, que directamente puede ser entendida por la CPU, sin necesidad de traductores ni compiladores. Las instrucciones son almacenadas a partir de una dirección de memoria y la ejecución de

ellas se realiza en forma secuencial. Supongamos que nuestra CPU soporta sólo cuatro operaciones básicas:

- *movimiento*: Mover el contenido de una posición de memoria a otra
- *suma*: Sumar el contenido de dos posiciones de memoria, dejando el resultado en la segunda.
- *comparación*: Comparar el contenido de dos posiciones de memoria
- *salto por igual a cero*: Salta a la posición destino si se cumple la condición, y si no sigue con la instrucción siguiente.

Obviamente nuestra CPU tiene un lenguaje de máquina muy limitado, pero nos sirve como ejemplo. Como tenemos cuatro operaciones, nos bastan 2 dígitos binarios para distinguir cada una de ellas. Entonces los dos primeros bits de nuestra instrucción sirven para que la CPU identifique cual operación tiene que realizar. Veamos un pequeño código en lenguaje de máquina:

00000111

01111000

10100111

11011000

A simple vista, no podríamos saber que significan estas cuatro instrucciones, pero si decimos que tomamos en cuenta la siguiente tabla para descifrar los códigos de operación, la cosa se simplifica notablemente:

OPERACIÓN	CODIGO ASOCIADO
suma	00
movimiento	01
comparación	10
salto	11

Cada CPU entiende el lenguaje de máquina diseñado para ella. Nuestra CPU entiende instrucciones de 8 bits, pero las instrucciones pueden ser de 16 ó 32 o incluso 64 bits, dependiendo de los diferentes modelos de CPU. Si tomamos la primera instrucción de nuestro programa anterior, podríamos establecer que, por ejemplo, los dos primeros bits (00) corresponden a la operación suma y, por la definición que dimos de la suma, sabremos que los siguientes tres bits forman la dirección del primer operando, y los últimos tres bits, la dirección del segundo operando, que además es el lugar donde se almacena la suma. Así podemos hacer el mismo ejercicio con las otras tres instrucciones.

Resulta obvio que para un programador es bastante tedioso realizar un programa medianamente complejo usando lenguaje de máquina. Tenemos varias desventajas:

1. Cada máquina tiene su propio lenguaje de máquina, con sus códigos de operaciones definidos, largo de instrucción, *etc.*, lo cual hace prácticamente imposible que un programador pueda trasladar sus programas de una máquina a otra.
2. Resulta bastante complicado encontrar un error entre tantos ceros y unos.
3. Programas simples, que en otros lenguajes corresponden a unas pocas líneas de código, en lenguaje de máquina tal vez ocupen hojas y hojas de ceros y unos.

Podríamos encontrar algunas desventajas más (dejo al lector la tarea de pensar en algunas).

Para aliviar la tarea del programador, cerca de 1950 surgen los *lenguajes ensambladores*. La característica principal de estos lenguajes, es que sus sentencias poseen una correspondencia uno a uno con las sentencias escritas en lenguaje de máquina. Simplemente se establece una notación simbólica para cada código de operación y los operandos, así como también para la representación de los datos. Si seguimos el ejemplo anterior, en lugar de decir 00 cada vez que queremos sumar, podemos decir ADD, entonces, podemos escribir nuestro programa anterior, del siguiente modo:

ADD A B -- Sumamos el contenido de la posición A al de B y se almacena en B

MOV B A -- Movemos el contenido de la posición B a la posición A

CMP C A -- Comparamos el contenido de la posición C con el de la posición A

JE D -- Saltamos a la posición D en caso que los contenidos de C y A sean iguales

¿Qué son A, B, C, D? Son etiquetas almacenadas en algún lugar de memoria que representan las direcciones: A=000, B=111, C=100, D=011.

Se ve claramente la correspondencia uno a uno con el lenguaje de máquina, y también se puede ver que para un programador es mucho más fácil de entender, depurar y modificar un código en lenguaje ensamblador que una serie de ceros y unos. De todos modos, aquí nos encontramos con un problema: la CPU no entiende sentencias en lenguaje ensamblador. Nuestro programa debe ser traducido a lenguaje de máquina. Un programa capaz de traducir las sentencias de un lenguaje ensamblador a un lenguaje de máquina, se llama, precisamente *ensamblador*.

A pesar de las facilidades que nos ofrece el lenguaje ensamblador, aún tenemos algunas desventajas:

- La programación insume aún mucho tiempo, dado que por cada instrucción en código máquina, tenemos una en lenguaje ensamblador, por lo tanto, tenemos programas muy largos y complejos.
- El lenguaje ensamblador aún es muy dependiente de la arquitectura de la máquina, por lo tanto un programa hecho en este lenguaje, es poco probable que podamos transportarlo a otra máquina.

Así, a mediados de la década del ’50, comenzaron a surgir los *lenguajes de alto nivel*. Este tipo de lenguajes es, en general, bastante independiente de la arquitectura de la máquina, es decir, no precisamos saber cuántos bits componen una instrucción en lenguaje de máquina, o cuantos registros tenemos disponibles, o en qué dirección de memoria se encuentra la primera instrucción a ejecutar, etc., etc. Podemos, en líneas generales, llevar de una máquina a otra, un programa hecho en un lenguaje de alto nivel, a veces sólo son necesarias pequeñas modificaciones para que funcione sin problemas. También es importante destacar que una instrucción en un lenguaje de alto nivel, es equivalente a un conjunto de instrucciones en lenguaje ensamblador o de máquina. Podemos entonces enumerar una serie de ventajas de este tipo de lenguajes:

1. Son más fáciles de aprender que los dos anteriores, dado que poseen una sintaxis que sólo se refiere a estructuras a utilizar, sin mención alguna a partes físicas de una computadora (como registros, memoria, etc.).
2. Los programas son más fáciles de elaborar, dado que requieren menos tiempo de escritura, son más cortos y concisos, son más fáciles de modificar y de encontrar y corregir errores.
3. Dada la independencia de la arquitectura de la máquina, podemos hacer programas más portables (es decir que se puedan trasladar de una máquina a otra).

Existen muchísimos lenguajes de alto nivel, nombraremos sólo algunos. Es importante comenzar la lista con el lenguaje **Fortran**, que surgió en 1954, creado por John Backus y patrocinado por IBM. Fue el primer lenguaje de alto nivel. A partir de allí, la lista comenzó a crecer rápidamente: **Algol**, **Cobol**, **Pl/1**, **Pascal**, **Ada**, **Snobol**, **Simula**, **Prolog**, **Lisp**, **C**, **Basic**, **Eiffel**, **Smalltalk**, **Java** y podríamos seguir varias hojas enumerando diversos lenguajes pertenecientes a diferentes paradigmas de programación.

Pues bien, volvemos a tener el mismo problema que con los lenguajes ensambladores: una computadora sólo entiende lenguaje de máquina, por ende precisamos de ‘algo’ capaz de traducir las sentencias escritas en un lenguaje de alto nivel, a lenguaje de máquina. Ese ‘algo’ recibe el nombre de *compilador*. Cuando escribimos un programa en un

lenguaje de alto nivel, utilizamos la sintaxis que previamente está definida para ese lenguaje. Por ejemplo, mientras que en **Eiffel**, una asignación se escribe:

```
a:= b+c;
```

en el lenguaje C#, la sentencia válida sería:

```
a= b+c;
```

es decir, existe un compilador por cada lenguaje y cada compilador conoce la sintaxis válida para ese lenguaje[1].

En síntesis, un *compilador* es un programa que recibe como entrada un conjunto de sentencias escritas en algún lenguaje de alto nivel y produce, como salida, un programa en lenguaje de máquina (normalmente denominado ejecutable) que puede ser ejecutado por la máquina donde fue compilado.

Para realizar esta compilación, se precisan de varias etapas. Dependiendo del lenguaje que se trate, esta traducción puede ser un proceso simple o puede ser bastante complejo. En términos muy generales, un compilador utiliza dos pasos para traducir el código fuente: en un primer paso, se analiza el código fuente, verificando la sintaxis y obteniendo información sobre, por ej., los nombres de las variables, entre otras cosas. En un segundo paso, con la información obtenida, se genera lo que se llama el código objeto. Una vez generado este código, puede ser que el mismo compilador (o a veces es un programa separado denominado *linker*) se encargue de la generación final del código ejecutable. Esta última etapa, se denomina etapa de vinculación y la idea es la siguiente: cuando el programa fuente está

[1] Eiffel tiene un motivo bien justificado para usar ":=" para la asignación en lugar de "=". La idea es que el símbolo de igual se usa justamente para la igualdad y que no es lo mismo que asignar un valor. La igualdad es una expresión que puede ser verdadera o falsa mientras que la asignación es una operación a realizar. El problema de usar = para asignación, además de ser incorrecto de leer, es que luego hay que usar otra cosa para la igualdad como por ejemplo el "= =" que usa C#.

compuesto de subprogramas que se compilan por separado, o bien utilizan librerías adicionales (o pre-compiladas), necesitamos unir todos esos 'trozos' de programas, que forman diferentes códigos objeto, y vincularlos a través del *linker* para, finalmente, obtener nuestro programa ejecutable. En síntesis, desde el momento en que el programador escribe su código en un lenguaje de alto nivel, hasta que obtiene un código ejecutable, se deben cumplir una serie de etapas:

1. La etapa de compilación, donde se traduce (en uno, dos o tres pasos, dependiendo de cada compilador) el código fuente a código objeto. Este código aún no se puede ejecutar.
2. La etapa de vinculación o *linkedición*, donde se toman todos los códigos objetos generados y se logra un código ejecutable y listo para ser utilizado.

Existe una variante al compilador, se trata del *intérprete*. En lugar de tomar un código fuente y traducirlo íntegramente a código ejecutable, el intérprete realiza la doble tarea de traducir y ejecutar simultáneamente, sin producir un código objeto ni un ejecutable. Lo que hace un intérprete es traducir cada línea del código fuente a medida que lo precisa, es decir al momento de ejecutarla. En general, la ejecución de un código interpretado es más lenta que la de un código compilado (¿Por qué?).

Dejo a cargo del lector, que descubra y enumere ventajas y desventajas de uno con respecto del otro.

Códigos ASCII y UNICODE

La información debe ser almacenada en la memoria de la computadora siguiendo algún estándar de codificación. La memoria de la computadora sólo almacena 1 y 0 en forma de número binarios. Naturalmente, los números binarios pueden ser escritos en formato hexadecimal o decimal. ¿Cuál es número que corresponde, por ejemplo, a la letra 'a'?. Eso depende de la codificación.

Cada uno o cero es lo que se denomina bit, ocho bits forman un byte. El código de representación más usado hasta ahora es el ASCII (también se utiliza el código EBCDIC).

ASCII significa **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange, mientras que EBCDIC significa **E**xtended **B**inary Code **D**ecimal **I**nterchange **C**ode. Si un programa escribe información en formato ASCII y otro que usa EBCDIC la lee, seguramente estamos en problemas.

Veamos la tabla ASCII:

ASCII	Hex	Símbolo
0	0	NUL
1	1	SOH
2	2	STX
3	3	ETX
4	4	EOT
5	5	ENQ
6	6	ACK
7	7	BEL
8	8	BS
9	9	TAB
10	A	LF
11	B	VT
12	C	FF
13	D	CR
14	E	SO
15	F	SI

ASCII	Hex	Símbolo
16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US

ASCII	Hex	Símbolo
32	20	(espacio)
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

ASCII	Hex	Símbolo
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

ASCII	Hex	Símbolo
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

ASCII	Hex	Símbolo
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

ASCII	Hex	Símbolo
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

ASCII	Hex	Símbolo
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	□

Cada carácter ocupa un byte (8 bits, donde un bit es un 0 o un 1), pudiendo encontrarse 256 caracteres distintos. El código ASCII normaliza los primeros 127 (aunque la extensión usa algunos números mayores). Los primeros 32 se llaman caracteres de control. Así, cuando guardamos un documento en formato ASCII, cuando se debe indicar que la línea ha terminado (CR carriage **return**) y que pase a la línea siguiente (LF line feed) se escriben los caracteres ASCII: 0A y 0D en Hexadecimal, naturalmente ya que en la computadora se guarda la representación binaria.

Actualmente existe un más moderno código de representación llamado UNICODE, el cual provee un único código para cada carácter independiente de la plataforma, el programa y el lenguaje. Si bien UNICODE está basado en ASCII (lo cual facilita la migración) evita el problema de ASCII que sólo cuenta con caracteres latinos. En ASCII es imposible representar palabras escritas, por ejemplo, en chino. Originalmente UNICODE contaba con una codificación en 2 bytes (en lugar de uno) lo que permitía codificar 65000 caracteres.

Actualmente UNICODE cuenta con tres formas de codificar la información que permite codificarla en 8, 16 o 32 bits. Las tres formas de codificar tienen el mismo repertorio de caracteres comunes y es posible cambiar una en otra sin pérdidas de datos. Las tres formas son referidas como UTF-8, UTF-16 y UTF-32

UTF-8: Usa una codificación de longitud variable, es el usado principalmente en páginas Web y tiene la ventaja de que coinciden los caracteres con su contraparte ASCII, es decir que, por ejemplo, la O mayúscula que en ASCII es el 4F, es el mismo en UTF-8.

UTF-16: Es útil ya que la mayoría de los caracteres utilizados se encuentran en una unidad de 16 bits, mientras el resto puede representarse en dos pares de unidades de 16 bits. Es bastante compacto.

UTF-32: Cada carácter ocupa 32 bits, puede ser usado cuando no necesitamos preocuparnos por el espacio.

De todas formas, el compromiso general es usar UTF-8 ya que es más compacto para la mayoría de las aplicaciones. Los lenguajes modernos como JAVA se basan en UNICODE.

Máquinas Virtuales

Si bien siempre es en última instancia el procesador el que ejecuta las instrucciones compiladas se ha desarrollado una alternativa a la compilación a código nativo: la utilización de máquinas virtuales.

Tanto Smalltalk, Java, como el marco de trabajo Microsoft .Net sobre el que se basa C# usan en alguna medida el concepto de máquina virtual. El problema que se intenta resolver con las máquinas virtuales es el de la portabilidad de las aplicaciones. Cada procesador y ambiente operativo es diferente por lo cual un compilador debería tomar en cuenta las diversas opciones para generar código específico. Así no sería lo mismo compilar para una computadora con un procesador de 64 bits que para una de 32 bits y a esto debería sumársele los diferentes sistemas operativos que brindan un ambiente de ejecución. Los compiladores en general producen un código que es de alguna manera el mínimo común denominador para que funcione en diferentes procesadores. De todas maneras portar un sistema de una plataforma a otra requiere, en la mayoría de los casos, volver a compilar el programa.

La otra alternativa es contar con un programa que simule ser una computadora genérica y que traslade algún código máquina (definido para dicha computadora virtual) al código máquina real donde se está ejecutando. De esta manera los compiladores sólo deberían preocuparse por generar el código adecuado para dicha máquina virtual y los programas compilados podrían ejecutarse en cualquier plataforma donde la máquina esté disponible. Este es el caso de .Net. El lenguaje C# fue diseñado específicamente para dicha plataforma. Otros lenguajes como Eiffel permiten generar código para dicha plataforma.
En realidad el marco de trabajo Microsoft .Net es más que una máquina virtual ya que además provee una completa librería de clases que facilitan la tarea del programador. Podemos decir que el marco de

trabajo .Net es una plataforma de programación y un conjunto de herramientas para construir aplicaciones.

Al compilar una aplicación C# (o Eiffel si así se desea) se genera código escrito en lo que se conoce como MSIL (lenguaje intermedio de Microsoft). Dicho código es trasladado a código nativo durante la ejecución (en un proceso conocido como compilación *just in time (jit).* El encargado de manejar la compilación *jit* es un componente del marco de trabajo .Net denominado CLR (C*ommon Language Runtime*).

Algoritmos y sentencias

Como veremos posteriormente, los algoritmos implementados estarán asociados a características de una clase de objetos. Pero antes de entrar en ese tema, veremos cómo se describe la secuencia ordenada de acciones que determina un algoritmo.

Dentro de la secuencia de acciones de un algoritmo, existen estructuras que nos permiten controlar que acción tomar en función del estado del programa y/o cálculos realizados sobre dicho estado. Lo que permite realizar dicha selección son las *estructuras de control.*

Hay tres tipos principales de estructuras de control:

- Secuencia
- Decisión o Selección
- Repetición o Iteración

Existe un cuarto tipo de instrucción de control que es la bifurcación o salto, la cual permite "saltar" desde una instrucción a otra en cualquier punto del programa si está debidamente etiquetada. Esto es muy común en lenguajes de bajo nivel como el lenguaje ensamblador, aunque también está presente en lenguajes de alto nivel, como la instrucción GOTO del Basic. No hablaremos en el resto del libro de esta instrucción. Su uso es considerado una mala práctica en

programación y se encuentra fuera de lo que se considera *programación estructurada* que es el estilo de programación que se realiza utilizando únicamente tres estructuras: secuencia, selección y repetición. Todo programa puede realizarse sin la utilización de la sentencia GOTO. El leguaje Eiffel no la tiene entre sus sentencias válidas. El padre de la programación estructurada, *Edsger W. Dijkstra*, escribió varios artículos al respecto, principalmente: "*A Case against the GO TO Statement*" originalmente publicado bajo el titulo "*Go To Statement Considered Harmful*" (la sentencia *goto* considerada dañina).

Definamos primero los que se denomina *flujo de control:*

Def*. El flujo de control de un algoritmo es el orden en el cual se deben ejecutar las acciones primitivas del mismo. Las estructuras que afectan al flujo de control se denominan estructuras de control.*

La *secuencia* determina un flujo de control que ejecuta una acción a continuación de otra según el orden en el que se encuentren escritas. Por ejemplo, el siguiente algoritmo en donde el flujo de control sigue desde la asignación de 1 a la variable a hasta la escritura de b en forma secuencial según el orden lexicográfico. Escribir una instrucción después de otra es lo que se denomina *secuencia.*

```
local
   a,b : INTEGER
do
  a := 1
  b:= 2
  a := a +b
  io.put_integer( a) -- escribo a en la pantalla
  io.put_integer( b) -- escribo b en la pantalla
end
```

No importa que no se comprenda en este momento todo el código anterior, lo que se quiere recalcar es que hay diversas sentencias, una a continuación de otra formando una secuencia que se ejecutará en el orden en que están escritas. Es importante recalcar que a veces los lenguajes exigen algún delimitador de línea para usar en las instrucciones, por ejemplo C# obligaría a poner un ";" al final de cada

línea (salvo en las estructuras de control) mientras que para Eiffel poner un ";" es optativo.

Decisión

La secuencia es una estructura trivial y sólo permite hacer programas simples que no necesiten modificar o cambiar el flujo de control. Algunas estructuras de control permiten repetir una cantidad de veces una secuencia de acciones, otras elegir entre dos o más acciones según determinadas condiciones. En rigor, sólo es necesario una estructura de repetición y una estructura de selección.

Pongamos, por ejemplo, que se necesite escribir cual de dos valores numéricos leídos es mayor. Un algoritmo, para hacerlo sería:

```
leer los valores
compararlos
escribir el mayor
```

Estamos escribiendo una visión general de los pasos a ejecutar, la estructura de decisión afectaría el segundo y tercer paso. En ellos debemos decidir cual valor es mayor y escribirlo.

Refinándolo, en Eiffel tendríamos:

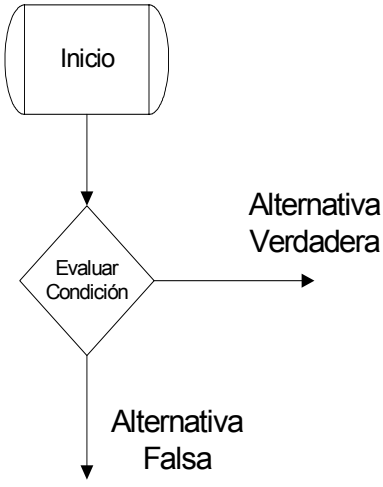
```
MostrarMayor is
local
    x,y: Entero
do
    io.read_integer(x)
io.read_integer(y)
if x >= y then
   io.put_integer(x) –escribir x
else
   io.put_integer(y)—escribir y
end
end
```

x >= y es una expresión lógica que describe una condición (el hecho de que x sea mayor o igual que y) y puede tomar dos valores:

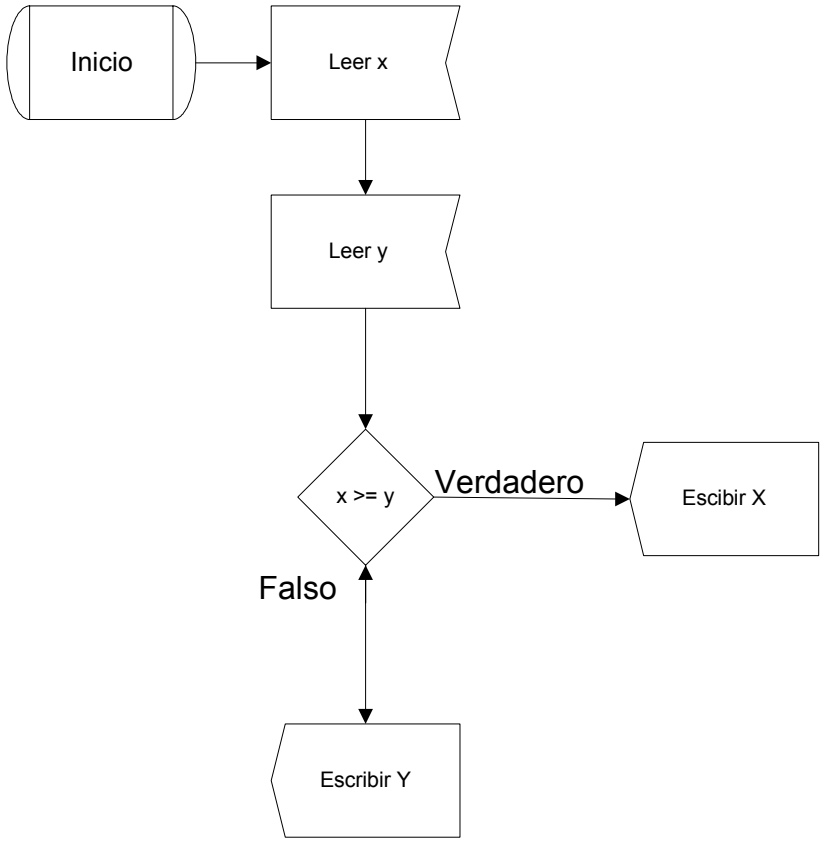
Verdadero o Falso. El algoritmo selecciona entre escribir `x` y escribir `y` en función del resultado de la evaluación de `x >= y`. Estamos en presencia de una estructura de control denominada selección y cuya forma constructiva es:

```
si <condición> entonces
        acciones si condición es verdadera
sino
        acciones si condición es falsa
finsi
```

Lo que expusimos recién es lo que se denomina *pseudocódigo*, es decir no es un lenguaje de programación sino que expresamos en español lo que cada lenguaje hace a su manera. El pseudocódigo es más informal que los lenguajes de programación pero dado un pseudocódigo es fácil trasladarlo a sentencias reales de un lenguaje. El si<condición> entonces es el **if** <condición> **then** del programa anterior. Vamos a especificar los ejemplos siguientes en pseudocódigo y luego mostraremos como las diversas sentencias se expresan en los diferentes lenguajes. La parte que viene a continuación de la palabra local es lo que denomina ambiente y permite definir las variables a usarse en el algoritmo. Una forma de especificar esta estructura es mediante lo que se denomina diagrama de flujo. Las técnicas modernas no lo usan ampliamente pero es más una herramienta de explicación y especificación que de construcción.



El rombo representa la decisión. Lamentablemente, en los diagramas de flujo no se visualiza tan claramente como en el pseudocódigo el grupo de acciones en cada alternativa, ya que si hay varias estructuras anidadas puede complicarse el diagrama. En nuestro ejemplo, el diagrama completo sería como la siguiente figura. En ella se ven los símbolos de lectura y escritura, el símbolo de inicio, etc.



En algunos casos, es posible que no exista alternativa falsa, es decir que sólo hay acciones que se ejecutan en caso de que la condición sea verdadera. La estructura es la siguiente:

```
si  <condición> entonces
        acciones si condición es verdadera
finsi
```

Veamos un ejemplo. Se trata de implementar un algoritmo que muestre la cantidad de horas que faltan para el fin de la jornada laboral, tomando en cuenta que los jueves se trabajan dos horas más y que la jornada tiene 5 horas. Se le pide al usuario que ingrese las horas trabajadas hasta el momento.

Método:

```
Pedir horas trabajadas y día de la semana
Calcular las horas faltantes, si el día es jueves sumar 2.
Mostrar horas faltantes
```

Refinamiento:

```
Algoritmo MostrarHoras
local
horas_trabajadas, horas_faltantes: Entero
Dia: Cadena de Caracteres
Comenzar
Leer horas_trabajadas
Leer Dia
horas_faltantes := 5 – horas_trabajadas
-- Chequear el dia de la semana
si dia = “Jueves” entonces
  horas_faltantes := horas_faltantes + 2
finsi
escribir horas_faltantes
fin algoritmo
```

Notemos la identación del algoritmo. Aquellas acciones que están dentro del alcance de las condiciones, se encuentran más a la derecha. Esto facilita enormemente la lectura y corrección. Todo algoritmo debe estar *identado*. Además, incorporamos un comentario que permite comprender mejor el algoritmo. Los comentarios en Eiffel, por ejemplo, son líneas precedidas por `--` . Cada lenguaje cuenta con su propios caracteres para delimitar comentarios: (*... *),{...), //.... , etc.

Operadores predefinidos y estándares en Eiffel

El lenguaje Eiffel tiene operadores predefinidos para la comparación. Son los siguientes: **=, /=, ~, /~.**

Los dos primeros expresan la igualdad y desigualdad de referencias y los dos segundos lo mismo, pero de objetos. Cuando veamos lo diferentes tipos de entidades (referencia y expandidos) se comprenderán las diferencias. Para valores numéricos como enteros y reales no tendremos por ahora mayores problemas en utilizar cualquiera de ellos.

Hay además un conjunto de operadores estándares que representan alias de funciones implementadas (eventualmente podrían ser redefinidas). Los mismos son: **+, –, *, /, ^, < >, <=, >=, //, \\,...** Su semántica es la esperada: suma, resta, multiplicación, división, potenciación, desigualdad, menor o igual, mayor o igual, división entera, resto y definición de intervalo. El más y el menos además son también operadores unarios para expresar, por ejemplo, números negativos o positivos.

Hay operadores lógicos como: not, and, or e implies, que representan la negación, la conjunción, la disyunción y la implicación lógica respectivamente.

Estructuras de decisión anidadas

Es posible anidar estructuras de decisión si se requiere. Veámoslo con un ejemplo: Se trata de realizar un algoritmo que determine el mayor de tres números. El algoritmo puede plantearse de la siguiente manera:

leer los 3 números
Calcular el mayor
mostrar el resultado

Podemos refinarlo más:

leer A, B, C. Si A es mayor que B entonces comparar A con C e informar el resultado, sino comparar B con C e informar el resultado. Un poco más explícito:

```
leer A, B, C
si A > B entonces
        comprara A con C e informar el resultado
sino
        comprar B con C e informar el resultado
finsi
```

Refinamos una vez más:

```
Algoritmo CalcularMaximodeTres
local
   A, B, C: Valores Numéricos
Comenzar
leer A, B, C
si A > B entonces
        si A > C entonces
                escribir 'El Máximo es: ', A
        sino
escribir 'El Máximo es: ', C
            finsi
sino
        si C > B entonces
                escribir 'El Máximo es: ', C
              sino
                escribir 'El Máximo es: ', B
      finsi
   finsi
fin algoritmo
```

Es importante ver como los distintos niveles de anidamiento nos muestran el alcance de las alternativas. Es fácil ver entonces la importancia de una buena identación. En este caso tenemos dos estructuras si-entonces-sino anidadas.

Estructuras de selección y lenguajes

Diversos lenguajes proveen diferentes formas de escribir (sintaxis) las estructuras de selección o decisión. Veamos algunos ejemplos:

Pascal:

```
if (condición) then
begin
        <alternativa verdadera>
end
else
begin
        <alternativa falsa>
end
```

Si las alternativas son sólo una línea:

```
if (condición) then
  <alternativa verdadera>
else
  <alternativa Falsa>;
```

En **C#** la estructura tiene la siguiente sintaxis:

```
if (<condición>)
{
  <alternativa verdadera>
}
else
{
  <alternativa falsa>
}
```

En este caso las llavecitas { y } son obligatorias. Pero si sólo hay una línea después del **if** o del **else** las llaves pueden omitirse. Notar que es importante en C# poner ";" al final de cada línea.

```
if ( 1 < actual)
    <una linea alternativa verdadera>;
else
  < una linea alternativa falsa> ;
```

En **Modula-2** (que excluye terminadores explícitos para las sentencias):

```
IF <condición> THEN
  <alternativa verdadera>
ELSE
  <alternativa falsa>
END
```

```
IF <condición> THEN
  <alternativa verdadera>
END
```

Las decisiones anidadas pueden escribirse en **Modula-2**:

```
IF condición THEN
        <sentencias1>
ELSEIF condicion2 THEN
  <sentencias2>
ELSEIF condicion3 THEN
  <sentencias3>
....
ELSE
        <sentencias>
END
```

Esto se lee de la siguiente manera: si se cumple condición entonces se ejecuta sentencias1 sino, si se cumple condicion2 se ejecutan sentencias2, sino si se cumple condicion3 se ejecutan sentencias3 sino, se ejecutan sentencias.

En **Fortran 77,** la sintaxis es similar a **Modula-2** excepto en las anidadas, donde ELSEIF se separa de la forma ELSE IF.

El lenguaje **Eiffel** tiene la siguiente sintaxis:

```
if <expresión boolena> then
 <instrucciones>
elseif <expresión boolena> then
  <instrucciones>
else
  <instrucciones>
end
```

El else final es optativo y puede haber un número arbitrario de cláusulas elseif que también son optativas.

Estructura de decisión generalizada o múltiple

Cuando es necesario ejecutar acciones dependiendo de muchos valores, a veces es conveniente tener una estructura que simplifique el anidamiento de la decisión.

```
si dia = 1 entonces
        ........
sino
     si dia = 2 entonces
                ..............
      sino
                si dia = 3 entonces
                ..............................
                sino
                        si dia = 4 entonces
                                ..............................
                        sino
                            si dia = 5 entonces
                                ..............
                             sino
                                        si dia = 6 entonces
                                                ......................
                                        sino
                                                ......................
                                         finsi
                                 finsi
                        finsi
                finsi
          finsi
finsi
```

En el código precedente se quieren realizar determinadas acciones dependiendo del día de la semana.

Claramente es bastante feo de leer y de escribir. Para evitar esto se utiliza la decisión múltiple. Su estructura es sencilla, como se ve en el código siguiente donde E representa una expresión y e1, e2.... representan expresiones o listas que se comparan con los resultados de la evaluación de la expresión E. El tipo de E depende del lenguaje, en general se soportan tipos enumerados, caracteres, conjuntos, etc. Los valores de comparación e1,...en. deben ser disjuntos para que funcione correctamente.

```
según E hacer
    e1: acciones1
        e2: acciones2
        e3: acciones3
de otro  modo
        acciones
finsegun
```

Utilizando la decisión múltiple el ejemplo de los días quedaría:

```
leer Dia
según Dia hacer
    1: acciones1
        2: acciones2
        3: acciones3
        4: acciones4
        5: acciones5
        6: acciones6
de otro  modo
        acciones del dia 7
finsegun
```

Vamos a hacer un ejemplo de utilización de la decisión múltiple con diversos valores. Supongamos que debemos hacer un algoritmo que calcula el monto de un cuota a pagar, según el día del mes que se paga. Si se paga el día 1 o 2 del mes se descuenta 3% si se paga hasta el día 5, no se cobra interés, si se paga del 5 al 10 se recarga un 10% del 10 al 20 un 12 % y después del día 20 un 15%. Veamos el algoritmo:

```
Algoritmo CalcularValorPago
local
    valorcuota: Real
    dia: Entero
Comenzar
  leer dia, valorcuota
  según dia hacer
   1,2: valorcuota := valorcuota – ( valorcutoa * 0.03)
   6..10: valorcuota := valorcuota + ( valorcutoa * 0.1)
   11..20: valorcuota := valorcuota + ( valorcutoa * 0.12)
   20..31: valorcuota := valorcuota + ( valorcutoa * 0.15)
  finsegun
  escribir valorcuota
fin algoritmo
```

En este caso, no utilizamos la parte `de otro modo` ya que es optativa y no la necesitamos. Si el día es 3 ,4 ó 5 no va a coincidir el valor de `dia` con ninguna rama, por lo cual no se ejecuta ninguna acción.

La primera rama tiene una lista de valores separados por comas. Si la expresión coincide con alguno de ellos entonces se ejecuta la acción correspondiente. El resto de las ramas trabaja con un rango definido por el límite inferior y el superior separados por dos puntos. Si la expresión cae dentro del rango, se ejecuta la acción correspondiente.

Algunos lenguajes como C no tienen esta estructura definida en forma de decisión múltiple sino parecida a un salto, dado que una vez que se encuentra la rama que coincide, se ejecutan todas las acciones del resto de las ramas de abajo. Esto obliga a poner una sentencia especial para evitarlo y saltar al final.

Repetición o Iteración

Una instrucción iteración permite englobar a una secuencia de instrucciones que se escribe una sola vez y se ejecuta un número de veces. Este tipo de instrucciones es llamado *bucle* y las instrucciones englobadas *cuerpo del bucle.* Usaremos tres tipos de instrucciones de

iteración, pero volvemos a aclarar que sólo una es necesaria. El lenguaje **Eiffel,** por ejemplo, sólo cuenta con una instrucción mientras que el **C#** cuenta con las tres que veremos.

Estructura repetir-hastaque

En general, puede no saberse la cantidad de veces que el *cuerpo del bucle* debe ejecutarse. En este caso se debe repetir el bucle hasta que se cumpla determinada condición. La estructura iterativa que usaremos es: **repetir-hastaque.** Tiene la siguiente forma:

```
repetir
        <cuerpo del bucle>
hastaque <condición de fin>
```

Supongamos por ejemplo la función factorial. El factorial de un número natural *n,* que se nota n!, se define como la siguiente función:

$$x! \begin{cases} 1 \text{ si } x = 0 \\ 1 \text{ si } x = 1 \\ x\text{-}1! \text{ si } x > 1 \end{cases}$$

Utilicemos la repetición para calcularlo:

```
Algoritmo Factorial
Local
   n , i, factorial: Entero
fin Ambiente
Comenzar
i := 1
factorial := 1
leer n
repetir
  factorial := factorial  * i
  i := i +1
hastaque  (i >= n +1)
escribir factorial
fin algoritmo
```

Estructura *mientras-hacer-finmientras*

Hay veces en las cuales es más fácil o claro expresar las condiciones que hacen que el bucle se ejecute, que determinar las condiciones de finalización del mismo. Es posible también que en algún caso no se deba ejecutar nunca el cuerpo del bucle (notar que en la estructura repetir-hastaque el cuerpo del bucle se ejecuta al menos una vez sin importar la condición). Para estos casos, usaremos la construcción mientras-hacer-finmientras**.** Tiene la siguiente estructura:

```
mientras <condición verdadera> hacer
  <cuerpo del bucle>
fin mientras
```

Un ejemplo muy interesante, es el de calcular el cociente entero entre dos números utilizando sólo la suma y la resta[2]. El método es ir restando el divisor al dividendo tantas veces como sea necesario. El cociente es la cantidad de restas realizadas y el resto es resultado de la última resta. Usaremos variables auxiliares para evitar modificar el valor original del dividendo por si luego lo queremos mostrar.

```
Algoritmo DividirEnteros
local
   dividendo, divisor, q,r: Enteros
Comenzar
leer dividendo, divisor
r := dividendo
q := 0
mientras  r >= divisor hacer
  r := r - divisor
  q := q + 1
finmientras

escribir  'El resultado de dividir ', dividendo, 'por', divisor, 'es:'q , 'con
resto:' r
fin algoritmo
```

[2] *Introducción a la programación y a las estructuras de Datos* - Alicia Gioia y Silvia Braunstein.

La sentencia finmientras actúa como delimitador del cuerpo del bucle. *Antes* de cada ejecución del cuerpo del bucle se evalúa el predicado r >= divisor, que cuando evalúa en falso determina el fin del bucle. De forma diferente, repetir-hastaque evalúa la condición *después* de cada ejecución del cuerpo del bucle.

Supongamos una implementación utilizando repetir-hastaque

```
Algoritmo DividirEnteros
local
   dividendo, divisor, q,r: Enteros
Comenzar
leer dividendo, divisor
r := dividendo
q := 0
repetir
  r := r - divisor
  q := q + 1
hastaque r < divisor
escribir  'El resultado de dividir ', dividendo, 'por', divisor, 'es:', q , 'con
resto:' , r
fin algoritmo
```

Notar que la condición es opuesta ya que el **repetir-hastaque** lleva la condición de fin y no la de continuar como hace **mientras-hacer-finmientras.**

Ambos algoritmos funcionan bien para muchos casos, pero el segundo no funciona para algunos casos ¿cuáles? Queda para el lector encontrar los casos en los cuales el segundo algoritmo no trabaja correctamente.

Estructura para-hacer-finpara

En ocasiones, se conoce previamente el total de iteraciones que se deben realizar en un bucle. Puede ser porque es un número fijo o un número ingresado por el usuario. Para el caso del factorial, el número de veces que se repite el cuerpo del bucle depende del valor de n. La estructura para-hacer-finpara es útil para estos casos. Su construcción es la siguiente:

```
para var desde vi hasta vf conpaso p hacer
  <cuerpo del bucle>
finpara
```

En esta estructura, `var` es una variable de tipo entero llamada *variable de control*, `vi` y `vf` son variables, constantes o expresiones de tipo entero; `vi` se denomina *valor inicial* y `vf` valor final. A la variable `p`, que puede ser un entero positivo o negativo pero distinto de 0, se le denomina *paso.* El paso es optativo, si no se incluye se asume un valor de 1.

El significado de esta estructura es el siguiente:

Asignar a var el valor de vi. Mientras var sea menor o igual que vf, ejecutar el cuerpo del bucle variando, al finalizar el cuerpo del bucle, el valor de var, sumándole la variable p.

Un equivalente usando mientras-hacer-finmientras sería, para paso positivo:

```
var := vi
mientras var <= vf hacer
   <cuerpo del bucle>
    var := var + p
finmientras
```

y el equivalente para paso negativo:

```
var := vi
mientras var >= vf hacer
   <cuerpo del bucle>
    var := var + p  --ojo! p es negativo
finmientras
```

Hagamos un ejemplo sencillo: Sumar los primeros 10 números naturales, para ello usaremos una variable i que vaya incrementándose desde 1 hasta 10 utilizando el ciclo para-hacer-finpara

```
Algoritmo Suma
local
      i,suma : Enteros
Comenzar
  suma:=0
  para i desde 1 hasta 10 hacer
     suma:=suma + i
  finpara
fin Algoritmo
```

Hagamos un segundo ejemplo para ver el uso del paso: Mostrar los 10 primero número naturales de mayor a menor

```
Algoritmo Mostrar
local
        i : Enteros
Comenzar
  para i desde 10 hasta 1 conpaso -1 hacer
     escribir  i
  finpara
fin Algoritmo
```

Estructuras de repetición y lenguajes

Hasta aquí vimos las estructuras de repetición en pseudo-código, ahora estudiaremos como se expresan dichas estructuras en los lenguajes de programación. Veremos las estructuras de repetición en C# y Eiffel.

En el lenguaje C#

```
while (<condición>)
{
         <instrucciones>
}
```

No hay un ***repetir-hastaque*** pero hay una forma de establecer que siempre se ejecute el cuerpo poniendo el while al final:

```
do
  <instrucciones>
while(<condición>);
```

La estructura **para-hacer-finpara** se escribe:

```
for (<inicialización>; <condición>; <modificación>)
                    <instrucciones>
```

Veamos un ejemplo:

```
for ( int vi= 0; vi  < 100; vi++)
{
        usar la variable vi..
}
```

En este ejemplo la variable vi es declarada *dentro* del for como entera, se itera hasta que vi sea igual a 100, la instrucción vi++ incrementa en uno a la variable vi. Notar que a diferencia del pseudocódigo anterior hay que incrementar explícitamente la variable.

En **Eiffel** sólo hay una estructura de ciclo, el loop, que es **suficiente** *para reemplazar a cualquiera de los iteradores*.

```
from
  <instrucciones de inicio>
until
  <condición de salida>
loop
  <cuerpo del bucle>
end
```

Ejemplo del factorial de N,

```
from
        i := 1
        factorial := 1
until
  i > N
loop
   factorial := factorial * i
   i := i+1
end
```

Ejercicios

1. Realizar un algoritmo que imprima el mínimo entre cuatro valores que se piden al usuario.
2. Realizar un programa que pida los valores de los tres lados de un triángulo e imprima si el mismo es equilátero, isósceles o escaleno. Hacer el diagrama de flujo correspondiente.
3. Leer tres valores que corresponden a un día, un mes y un año y determinar si se trata de una fecha válida o no.
4. Modificar el ejercicio 3) para que, el programa itere hasta que se ingrese una fecha válida. Es decir, mientras que la fecha sea inválida, seguir en el ciclo.
5. Hacer un programa que imprima la suma de los 30 primeros números pares. Realizar dos versiones del mismo programa: una que use la estructura **mientras-hacer-finmientras** y otra versión que use **para-hacer-finpara**

Primeros Pasos con Objetos

¿Que son los Objetos?

La noción de objetos aparece por primera vez en la década de 1960, cuando un grupo de científicos comenzó a pensar un modelo de programación totalmente diferente a los existentes hasta entonces y que podía ser útil en el área de simulación. La idea fue diseñar un lenguaje que permita describir directamente los objetos que serían simulados. Estos conceptos fueron volcados en el lenguaje **Simula**, considerado como el primer lenguaje orientado a objetos. Luego descubrieron que el concepto de objetos podía ser aplicado en algo más que en simulación.

Simula ha influenciado el diseño de muchos otros lenguajes orientados a objetos. El primero de ellos fue **Smalltalk**, que surge en los años ’70. Luego, más adelante, a partir de la década de los ’80 y hasta ahora aparecen muchos otros lenguajes orientados a objetos: **C++**, **Eiffel**, **Objective C**, **Object Pascal**, **Modula-3**, **Ada95**, **Java**, **C#,** etc. Con el *boom* de los objetos, ya finalizando la década de los ’80, aparecen técnicas de análisis y diseño orientadas a objetos en el área de ingeniería de software que actualmente son de gran uso en aplicaciones comerciales e industriales.

Hoy en día existe aún una gran confusión respecto de lo que es la programación orientada a objetos y de cuáles serían las características necesarias para que un lenguaje pueda ser considerado orientado a objetos. Peter Wegner, en laconferencia de la OSPLA (Conference on Object-Oriented Programming Systems, Languages, and Applications) de 1987, presentó un artículo titulado: *Dimensions of Object-Based Language Design* en el cual se presenta una categorización de los lenguajes según algunas características claves que estos posean: *objetos, clases, herencia, abstracción, tipificación y concurrencia.*

Parte de dicha clasificación es la siguiente:

1. *Lenguajes basados en objetos*: lenguajes que soportan objetos como característica propia del lenguaje. Los objetos sirven para agrupar operaciones y datos y facilitan el diseño de programas orientados a datos.
2. *Lenguajes basados en clases*: lenguajes en los cuales cada objeto pertenece a una clase pero sin un mecanismo para la organización de clases. Las clases sirven para manejar colecciones de objetos, permitiendo ver a los mismos como valores de una clase dentro del lenguaje. De esta forma, los objetos pueden ser pasados como parámetros, asignarlos como valores a variables y organizarlos en estructuras.
3. *Lenguajes orientados a objetos*: lenguajes en los cuales cada objeto pertenece a una clase y donde es posible la definición incremental de jerarquías de clase a través de un mecanismo de herencia. La herencia es utilizada como una forma para compartir recursos y permite describir los dominios de las aplicaciones.

Cabe aclarar que cada una de estas categorías está incluida en la anterior. Es decir, si un lenguaje es orientado a objetos, entonces también es basado en clases y es basado en objetos. Un lenguaje basado en clases, es también basado en objetos pero no es orientado a objetos. A medida que se avance se irán comprendiendo estos conceptos.

En programación orientada a objetos, un programa es una colección de objetos que trabajan y cooperan entre sí. Un objeto es básicamente una máquina que almacena una colección de datos y expone un comportamiento. El programa manipula estas máquinas mediante dos tipos básicos de operaciones: consultas y comandos. Las consultas nos permiten tener acceso a los datos que la máquina sostiene y los comandos nos permiten modificar dichos datos.

El principal objetivo es solucionar la complejidad existente en los problemas del mundo real mediante la abstracción y el encapsulamiento del conocimiento en objetos.