FUNDAMENTOS DE PROGRAMACIÓN

Tema I: Teoría Semántica de la Lógica Matemática. Introducción a la Teoría semántica del cálculo proposicional.

Introducción a la Lógica Matemática.

La lógica es la ciencia que estudia el proceso de razonamiento, particularmente enfocado al análisis sobre si un determinado razonamiento es correcto o no.
La lógica como ciencia se centra en el estudio de la estructura del razonamiento, por ejemplo, para saber cuales son las premisas necesarias para arribar a una conclusión. El significado o utilidad particular de esa conclusión dependerá de la rama de la ciencia o de la vida en cuestión.
La Lógica es una de las ciencias más antiguas y los griegos desde Sócrates, Platón y Aristóteles le dedicaron grandes esfuerzos a desarrollarla. Su desarrollo también estuvo mucho tiempo ligado al arte de la discusión y la oratoria, ya que la Lógica brinda instrumentos muy útiles para demostrar o refutar juicios.
Desde aquellos tiempos, los pensadores se dedicaron a formalizar dentro de sus posibilidades las estructuras correctas de razonamiento. En los siglos recientes, la Lógica ha adquirido un carácter mucho más formal y matemático, permitiendo aumentar notablemente su utilidad para el avance de todas las ciencias.
Dentro de la informática, la Lógica se usa para demostrar que hacen realmente los programas, y por eso este folleto brindará los elementos esenciales de la Lógica Matemática para luego dar paso a una introducción a los algoritmos y a la programación.
La Lógica Matemática clásica, que es el objeto de estudio de esta primera parte, se divide ***de acuerdo a los elementos básicos que utiliza para representar enunciados o frases del mundo real*** en dos partes que son las siguientes:
Lógica de Proposiciones (Proposicional)
Lógica de Predicados.
Por su parte, ***según la manera de justificar la validez de los razonamientos***, la Lógica se divide en:
Teoría Semántica
Teoría Sintáctica.

Proposiciones simples

Una proposición o enunciado es una oración que puede ser verdadera o falsa pero no ambas a la vez. O sea, la principal propiedad de una proposición es que toma uno de valores de verdad posibles, o bien son verdaderas o bien son falsas. Aquí por ejemplo no se incluyen las frases que sean preguntas, instrucciones, órdenes, etc. Esto es así ya que la lógica como ciencia persigue el estudio del proceso de razonamiento y debe centrarse en las frases que formen parte de un proceso de razonamiento. En este caso se incluyen las frases de tipo declarativo que expresan relaciones entre sujetos, aseveraciones de que determinadas propiedades se cumplen o no, etc.
La proposición es un elemento fundamental de la lógica matemática que trabaja solamente con proposiciones o Lógica Proposicional o de Proposiciones (también recibe el nombre de Cálculo Proposicional). Por eso, lo primero es reconocer que oraciones o frases constituyen proposiciones y cuales no.
A continuación se tienen algunos ejemplos de frases. Sin embargo, mientras que algunas de ellas son proposiciones otras no lo son. Las frases se indican por medio de una letra minúscula, dos

puntos y la proposición propiamente dicha. Esta es la notación que se usará normalmente en el folleto. Las frases que analizaremos son las siguientes.

p : El sol gira alrededor de la tierra.
q : -17 + 38 = 21
r : x > y-9
s : Holguín ganará el campeonato nacional de béisbol del año 2003.
t : Hola ¿como estas?
u : Al combate, corred, bayameses.
v : ¿Que vadis? ¿a dónde vas?
w : "Condenadme", no importa.

Los incisos p y q sabemos que pueden tomar un valor de verdadero o falso; por lo tanto son proposiciones. En el caso de la proposición p sabemos que toma un valor falso.
Por su parte, la proposición q es verdadera y quien lo dude puede calcularlo.
El inciso r también es una proposición, aunque el valor de falso o verdadero depende del valor asignado a las variables "x" y "y" en determinado momento. Algo similar ocurre con la proposición del inciso s que también está perfectamente expresada aunque para decir si es falsa o verdadera se tendría que esperar a que terminara la temporada próxima del béisbol nacional.
Sin embargo los enunciados t, u y v no son proposiciones, ya que no pueden tomar un valor de falso o verdadero, el primero de ellos es un saludo, el segundo es una orden, arenga o solicitud, el tercero es una pregunta.
Sin embargo, no deben analizarse las frases de una manera mecánica (y esto es casi una ley para todo aquel que se adentre en los caminos de la informática) pues en ocasiones aunque la frase contenga una orden o indicación puede ser una proposición. Un ejemplo de esto lo encontramos en un fragmento de la famosa frase de Fidel en el juicio del Moncada que aparece como frase w. Esta sí es una proposición, porque la frase no es una orden en sí, aunque contiene una referencia a una frase que sí lo sea. De hecho, la frase se debe interpretar como "no importa que me condenen" lo cual sin duda alguna es una proposición.

Proposiciones compuestas.
Las frases vistas en la sección anterior son proposiciones simples. Sin embargo, existen otras proposiciones más complejas que no están solamente compuestas por una proposición simple, a estas se les llama proposiciones compuestas. Ejemplos de estas proposiciones compuestas son las siguientes:

p : Dos y dos son cuatro; cuatro y dos son seis
q : 4<6 y 6<8
r Hay sol bueno, mar de espuma, arena fina y Pilar quiere salir a estrenar su sombrerito de plumas.
s : No existe en la historia de las Series Nacionales de Béisbol una final tan disputada como la de este año 2002.
t : La final de la Copa del Mundo de Fútbol 2002 la ganó Brasil y no defraudó a su afición.
u : El número 8 es par o es impar.

Lo primero que puede verificarse es que en todos los casos se trata de proposiciones pues todas pueden ser verdaderas o falsas. La diferencia fundamental respecto a las proposiciones vistas en la sección anterior, es que estas contienen proposiciones más simples en su interior.
La primera proposición p contiene dos proposiciones más simples que son la proposición "Dos y dos son cuatro" y la proposición "cuatro y dos son seis". De manera similar, la proposición q está compuesta por las proposiciones simples "4<6" y por la proposición "6<8". La siguiente proposición r tomada del famoso poema "Los zapaticos de rosa" de Martí en un principio se ve que está formada al menos por dos proposiciones más simples. La primera es "Hay sol bueno, mar de espuma, arena fina" y la segunda proposición sería "Pilar quiere salir a estrenar su sombrerito de plumas". Sin embargo, puede considerarse que la primera de estas proposiciones aún no es simple, ya que está compuesta por tres proposiciones que serían "Hay sol bueno", "hay mar de espuma" y "hay arena fina". Estas tres proposiciones no se ven a simple vista porque no están explícitas debido al uso de la coma (",") pero sin duda este es el sentido de la frase.
La siguiente proposición s contiene en su interior la proposición "existe en la historia de las Series Nacionales de Béisbol una final tan disputada como la de este año 2002". Sin embargo la proposición s es la negación de esta.
La proposición t contiene las proposiciones "La final de la Copa del Mundo de Fútbol 2002 la ganó Brasil" y la proposición "defraudó a su afición", lo que en el caso de la segunda aparece negada.
Por último, la proposición u está compuesta por las proposiciones "El número 8 es par" y la proposición "el número 8 es impar".
Es importante notar que estas proposiciones simples que hemos identificado se unen para formar las proposiciones más complejas por medio de conectivas lógicas. En los ejemplos vistos anteriormente las conectivas lógicas que se han usado son la conjunción, la disyunción y la negación.
Veremos ahora estas conectivas para luego regresar sobre los ejemplos anteriores.

Conectivas lógicas.

Existen conectivas u operadores lógicos que permiten formar proposiciones compuestas (formadas por varias proposiciones). Los operadores o conectivas básicos son:

Operador de conjunción.

El operador de conjunción se utiliza para conectar dos proposiciones que se deben cumplir para que se pueda obtener un resultado verdadero. O sea, la proposición compuesta que contiene una conjunción está afirmando que se cumplen las proposiciones más simples que la componen.
Se representa usando el símbolo $\wedge$. Se le conoce también como multiplicación lógica, operador "and", operador "y". Otros símbolos con los que se representa son "." y "$\cap$".

Por ejemplo, la proposición "4<6 y 6<8" vista anteriormente es una conjunción. Sean:

w : 4<6
x : 6<8

Entonces la proposición compuesta anterior usando simbología lógica queda como sigue:

$q = w \wedge x$

El operador de conjunción en el idioma español aparece de diversas maneras pero la más usual es que sea a través de la palabra "y", tal es el caso de las proposiciones q, r y t. Sin embargo en ocasiones, puede ser a través de una coma como aparece es las proposiciones p y r anteriores. En el caso de la proposición r, ésta está compuesta por varias proposiciones simples en conjunción que serían:

w : Hay sol bueno
x : Hay mar de espuma
y : Hay arena fina
z : Pilar quiere salir a estrenar su sobrerito de plumas

Esta proposición r quedaría:

$r = w \wedge x \wedge y \wedge z$

Algunas formas en español para denotar una conjunción son las expresiones:

- **p** y **q**
- **p** pero **q**
- **p** sin embargo **q** (como la famosa "sin embargo se mueve" de Galileo en adición a cuando lo obligaron a darle la razón a los que afirmaban que la tierra no se movía)
- **p** no obstante **q**
- **p** a pesar de **q**
- **p** además de **q**
- **p** en adición a **q**

Observe que existe una diferencia entre frases como "y", "además" y "en adición" que significan que ambas afirmaciones están en línea o consonancia, en contraposición a lo que ocurre en frases como "pero", "no obstante" y "aunque" que expresan que las frases van en sentido opuesto o se contradicen de cierta manera. A pesar de esta diferencia que el lenguaje español logra expresar, para el lenguaje de la Lógica ambas de expresan con la misma conectiva $\wedge$ ya que la Lógica como ciencia no se preocupa por el contenido de cada proposición en sí y por tanto para ella estas diferencias no se expresan.

Operador de disyunción.
Con el operador de disyunción se obtiene un resultado verdadero cuando alguna de las proposiciones es verdadera. Se indica generalmente por medio del símbolo $\vee$. Se conoce también como suma lógica, operador "or", operador "o". Otros símbolos también usados para representarlo son "+" y "$\cup$".
Por ejemplo, la proposición **u** anterior "El número 8 es par o es impar" es una disyunción.

w : El número 8 es par.
x : El número 8 es impar.

Entonces la proposición compuesta anterior usando simbología lógica queda como sigue:

$u = w \vee x$

En español existen varias frases para denotar disyunciones entre las que están:

- o, u
- o **p** o **q** o ambas cosas
- al menos **p** o **q**
- como mínimo **p** o **q**

Existe otro operador de disyunción llamado disyunción exclusiva o "xor", cuyo funcionamiento es semejante al operador de disyunción con la diferencia que su resultado es verdadero solamente si una de las proposiciones es cierta, cuando ambas proposiciones son verdaderas entonces el resultado es falso. El uso de este operador es más limitado por eso no insistiremos en él por el momento.

Operador de negación.

La función del operador de negación es negar otra proposición. Esto significa que si alguna proposición es verdadera y se le aplica el operador de negación se obtendrá su complemento o negación (falso). Este operador se indica por medio de los siguientes símbolos: ‘, ¬, –, ~. En el libro de texto de la signatura se indica generalmente por medio de una barra sobre la proposición. Aquí usaremos el símbolo – delante. Por ejemplo, la negación de la proposición p se representa como -p.
Por ejemplo, si se cumple que:

w : Existe en la historia series nacionales de béisbol una final tan disputada como la de este año 2002.

Entonces la proposición compuesta **s** vista anteriormente usando simbología lógica queda:

$s = -w$

En español algunas de las frases más usadas para denotar la negación de una proposición **p** cualquiera son:

- no **p**
- es falso que **p**
- no es cierto que **p**
- no se cumple que **p**

También se encuentran negaciones dentro de proposiciones con otras conectivas pues las proposiciones compuestas pueden contener más de una conectiva lógica. Por ejemplo, la proposición **t** anterior "La final de la Copa del Mundo de Fútbol 2002 la ganó Brasil y no defraudó a su afición" contiene una conjunción y una negación.
Siendo:

w : La final de la Copa del Mundo de Fútbol 2002 la ganó Brasil

x : defraudó a su afición

Entonces la proposición compuesta t anterior usando simbología lógica quedaría:

$t = w \wedge -x$

Notación de la lógica proposicional.

En la sección anterior se presentaron algunos de los operadores lógicos más importantes y la notación que se usa para representarlos. Aquí sólo se pretende hacer un resumen de la notación que se usará en el resto del curso para representar de manera formal los mismos.
Para representar a las proposiciones se usarán las letras minúsculas del alfabeto empezando generalmente por las letras p, q, r... z. En ocasiones es necesario representar más proposiciones. Para estos casos se podrás usar otras letras cualesquiera del alfabeto, pero siempre en minúsculas.
Cuando se vaya a representar el significado de una proposición simbolizada por una letra, se debe hacer poniendo primero la letra que se usará, luego dos puntos ":" y posteriormente la frase en idioma español. Por ejemplo:

w : Hoy he aprendido algunas cosas nuevas.

Los símbolos usados para representar cada operador o conectiva son:
Conjunción $\wedge$
Disyunción $\vee$
Negación – (en el caso del libro de texto, el símbolo usado es una barra sobre la letra que simboliza a la proposición que se negará)
En el caso de que sea necesario agrupar proposiciones se pueden usar paréntesis. Esto es muy importante porque es incorrecto, por ejemplo, escribir $p \wedge q \vee r$ ya que no queda claro el significado. Suponga que

p : le gusta el voleibol

q : le gusta el fútbol

r : le gusta el béisbol

Existen dos interpretaciones, la primera se correspondería con $(p \wedge q) \vee r$ lo que significaría que a esa persona "le gusta el voleibol y el fútbol, o le gusta el béisbol". Sin embargo, si se interpreta como $p \wedge (q \vee r)$ lo que significaría es que a esa persona "le gusta el voleibol y también le gusta el fútbol o el béisbol". Por ejemplo, para una persona que solamente el béisbol la primera lectura es "verdadera" (V), sin embargo la segunda es "falsa" (F) ya que no se cumple p que está en conjunción con la disyunción de las otras. En casos como estos donde pueden haber varias lecturas los paréntesis son necesarios. En los otros pueden usarse o no para lograr más claridad.

En cuanto a la representación de los valores de verdad, se usa general mente la letra V mayúscula para representar el valor "verdadero" y la letra F mayúscula para representar el valor "falso".
Introducción a la Teoría semántica del cálculo proposicional

La Teoría semántica

Según Cuena en su libro Lógica Informática (Parte I, pag. 83), la Teoría Semántica como sistema lógico para saber la validez de los razonamientos se construye "mediante una simbolización del **significado** de las proposiciones, es decir, de la forma de valorar el **contenido** de información de cada proposición". Esta es la razón por la que se le llama "semántica" ya que esta palabra se refiere al significado o información.
Este sistema se basa en varios elementos esenciales entre los que se destaca:
Un conjunto de significados atribuibles a las proposiciones.
Una definición semántica de conectivas, es decir, de la forma con que se atribuye significado a las proposiciones compuestas construidas con los distintos tipos de conectivas, a partir del significado de las proposiciones que lo componen.

Significados o valores de verdad de las proposiciones

A partir de lo visto antes, lo primero que debemos definir son los valores de verdad que pueden tomar las proposiciones. Como se ha venido viendo, cada proposición puede tomar un valor de verdad verdadero (V) o falso (F). Por tanto ya tenemos el primer elemento quedando definido el conjunto de significados atribuibles a las proposiciones como {V, F}.
Este par de significados es suficiente en la mayoría de los casos y será el que veremos durante el curso. Sin embargo es bueno comentar que el desarrollo actual de la ciencia ha llevado a ampliar estos conceptos. En algunos sistemas lógicos como es la Lógica Trivalente se admiten los valores "verdadero", "falso" y "desconocido", mientras que en la Lógica Modal se trabaja con los valores "necesario" y "posible". A estas lógicas se les llaman lógicas no clásicas. Otro problema que se trata de enfrentar con la asignación de otros valores de verdad es poder razonar con frases como "Esta frase que está leyendo es falsa" que evidentemente provoca un círculo vicioso y cuesta trabajo determinar que debe hacerse con la misma. Resumiendo, se asumirá como valores de verdad para las proposiciones el conjunto {V, F}.

Definición semántica de las conectivas

El otro elemento necesario para conformar la Teoría Semántica es la definición semántica de las conectivas, o sea, que cuando sepamos el valor de las proposiciones que componen una proposición compuesta podamos obtener el valor de esta.
Por ejemplo, sabemos que los valores de verdad de determinadas proposiciones son los siguientes: p = F, q = F, r = V, y queremos saber que valor V o F toma la proposición compuesta $p \wedge (q \vee r)$. Si sustituimos entonces estamos preguntándonos el valor de verdad de $F \wedge (F \vee V)$. Para resolver esto tenemos que primero saber que valor de verdad debe tener $(F \vee V)$. De manera intuitiva seguro se piensa que si la disyunción de una proposición verdadera (V) y una falsa (F) debe valer verdadero (V). Si es así, entonces lo que nos queda es saber qué valor de verdad debe tomar $F \wedge (V)$ como se trata de la conjunción de una proposición verdadera y una falsa, entonces también de manera intuitiva seguro se piensa que debe ser valer falso F el resultado final. De esta manera, se ha obtenido el valor de la proposición compuestas deseada $p \wedge (q \vee r)$ para la interpretación p = F, q = F, r = V. Resumiendo los pasos tenemos:

p $\wedge$ **(q** $\vee$ **r)** con p = F, q = F, r = V
F $\wedge$ (F $\vee$ V)
F $\wedge$ (V)
F
De esta forma se van construyendo los valores de verdad de cualquier proposición por compleja que esta sea. Aquí hemos visto de un modo intuitivo la obtención de valores de verdad para proposiciones compuestas, sin embargo, esto debe quedar de manera precisa.
O sea, la Teoría Semántica debe definir qué valores de verdad debe tomar cada proposición compuesta para cada uno de los valores de verdad que puedan tomar las proposiciones simples que la forman.
Para hacer esto se utilizan las tablas de verdad, las cuales expresan el significado de cada proposición compuesta en cada caso.

Tablas de verdad de conectivas (negación, conjunción y disyunción).

Las tablas de verdad se usan para representar los valores de verdad de la aplicación de los operadores o conectivas a proposiciones más simples. En las tablas de verdad se representa de manera explícita todas las combinaciones de valores de las proposiciones simples que contiene la proposición compuesta y para cada uno de ellos se coloca el valor de verdad resultante de aplicar el operador dado. De esta manera es que se realiza la definición semántica de las conectivas.
Para las conectivas básicas las tablas de verdad se asumen a partir del significado usual de las mismas. Para proposiciones que integran varias, el resultado se obtiene aplicando sucesivamente los operadores.
El valor "verdadero" se representa con una V y el valor "falso" con una F.
Para las conectivas u operadores básicos, las tablas de verdad son las siguientes.

Negación	
p	-p
V	F
F	V

Esta tabla de verdad se interpreta como:
Cuando p es V, entonces –p es F
Cuando p es F, entonces –p es V

Como se puede ver, siempre la proposición -p tiene el valor contrario al que tiene p.

Conjunción		
p	q	p∧q
V	V	V
V	F	F
F	V	F
F	F	F

Esta tabla de verdad se interpreta como:
Cuando tanto p como q son V, entonces p∧q es V
En el resto de los casos p∧q es F

Esto se corresponde con la lectura que le damos normalmente al operador de conjunción. Por ejemplo, si una persona dice: "ella habla inglés y además habla alemán", entonces esta proposición compuesta sólo es verdadera (V) si ambas proposiciones simples son también verdaderas. Si la persona realmente no habla alguno de los dos idiomas (o sea, que sea falso "ella habla inglés" o que sea falso "habla alemán") entonces la proposición compuesta es falsa (F).

Disyunción		
p	q	p∨q
V	V	V
V	F	V
F	V	V
F	F	F

Esta tabla de verdad se interpreta como:
Cuando tanto p como q son F, entonces p∨q es F
En el resto de los casos p∨q es V

Esto se corresponde con la lectura que le damos normalmente al operador de disyunción. Por ejemplo, si una persona dice: "ella habla inglés o habla alemán", entonces esta proposición compuesta sólo es falsa (F) si ambas proposiciones simples son también falsas. Si la persona realmente habla alguno de los dos idiomas (o sea, que sea verdadero "ella habla inglés" o que sea verdadero "habla alemán") entonces esto bastaría para que la proposición compuesta sea verdadera (V).
Para el caso de la disyunción exclusiva, la diferencia radica en que el primer caso (donde ambas proposiciones son verdaderas) el valor de verdad de la disyunción exclusiva es falso. O sea, esta operación exige que una y solo una de las proposiciones se cumpla. Este operador tiene una gran utilidad en la electrónica digital ya que evalúa que ambos valores de verdad sean diferentes, pero en la Lógica se usa realmente muy poco.

Interpretaciones, modelos y contramodelos

Cada una de las filas de las tablas de verdad se corresponde con un juego de valores de verdad para cada una de las proposiciones simples. A cada una de estos juegos de valores que se corresponde con la asignación de valores a las proposiciones simples componentes, y que se corresponde además con una cada fila de la tabla de verdad se le llama interpretación.
Para cada una de las interpretaciones, la proposición compuesta para la que se está haciendo su tabla de verdad toma un valor de verdad que como se ha visto puede ser "verdadero" (V) o "falso" (F). Las interpretaciones con valor V reciben el nombre de modelos o ejemplos de la proposición analizada. Por otra parte, las interpretaciones con valor F reciben el nombre de contramodelos o contraejemplos de la proposición analizada.

En las tablas de verdad anteriores podemos ver que:
La conectiva lógica de negación tiene dos interpretaciones que son p =V y p = F. La primera de ellas es un contramodelos y la segunda es un modelo.
La conectiva de conjunción tiene cuatro interpretaciones que son (p= V, q = V), (p=V, q = F), (p = F, q =V) y (p = F, q = F). De estas solamente (p = V, q =V) es modelo mientras que las otras tres son contramodelos.
La conectiva de disyunción también tiene cuatro interpretaciones que al igual que para la conjunción son (p = V, q = V), (p = V, q = F), (p = F, q =V) y (p = F, q = F). De estas solamente (p = F, q = F) es contramodelos mientras que las otras tres son modelos.

Tablas de verdad de proposiciones compuestas

Preparación de la tabla de verdad para cualquier número de variables.

Después de ver las definiciones de cada una de las tablas de verdad para las conectivas, es posible crear tablas de verdad para proposiciones más complejas. La manera de hacerlo es muy simple y consiste en la aplicación paso a paso de las tablas de verdad para las conectivas según sea el caso de la proposición compuesta a la que se le quiere hacer la tabla de verdad. Veamos algunos ejemplos.
Si se quiere hacer la tabla de verdad de la proposición $p \wedge q \wedge r$ lo primero que tenemos que identificar es el número de proposiciones que la componen. En este caso son tres proposiciones por tanto la cantidad de filas de la tabla de verdad sería la cantidad de combinaciones de las 3 variables. En este caso son ocho filas ya que la cantidad de combinaciones es $8 = 2 \cdot 2 \cdot 2 = 2^3$. En general, la cantidad de filas es 2^n, siendo n la cantidad de variables. A cada fila se le llama interpretación y se corresponde con la asignación de un valor de verdad para proposición simple.
Para la construcción de la tabla de verdad debemos ser cuidadosos, no repitiendo las combinaciones. La manera usual de lograr esto es fijando un valor para la primera variable y ver las combinaciones del resto de las variables con ella. Luego se repite estos dos pasos para obtener las combinaciones del resto de las variables. Por ejemplo, si hay una sola variable como es el caso de la tabla de verdad de la negación entonces es muy sencillo: primero V y luego F. Quedando la tabla de verdad cuando hay una sola variable como:

p	…
V	...
F	…

Si fueran dos variables p y q, partimos de dejar fija la primera variable (p) al primer valor (V) y para este valor generar las dos combinaciones de la otra variable. Luego, se debe hacer igual para el otro valor de la primera variable (p). La tabla de verdad quedaría así.

p	q	...
V	V	...
V	F	...
F	V	...
F	F	...

Puede observarse que las dos primeras filas corresponden a las combinaciones que tienen fijo del valor V para la primera variable p, luego la tercera y cuarta filas se corresponden a los casos en que la primera variable es toma valor F.
Continuando esta línea para tres variables se fija primero el valor V para la primera variable y se colocan todas las combinaciones del resto de las variables con este valor de verdad para la primera. Luego se fija el valor F para la primera variable y ponen de nuevo todas las combinaciones del resto de las variables. Para tres variables la tabla de verdad quedaría:

p	q	r	...
V	V	V	...
V	V	F	...
V	F	V	...
V	F	F	...
F	V	V	
F	V	F	
F	F	V	
F	F	F	

Para cuatro variables sería:

p	q	r	s	...
V	V	V	V	...
V	V	V	F	...
V	V	F	V	...
V	V	F	F	...
V	F	V	V	...
V	F	V	F	...
V	F	F	V	...
V	F	F	F	...
F	V	V	V	...
F	V	V	F	...
F	V	F	V	...
F	V	F	F	...
F	F	V	V	...
F	F	V	F	...
F	F	F	V	...
F	F	F	F	...

- Otra manera de formar las tablas de verdad para que queden de la manera anterior es:
- empezar por la última variable y poner UN valor V y uno F alternadamente
- si hay alguna variable aún sin valores, ir a la penúltima variable y poner DOS valores V y luego DOS valores F alternadamente
- si hay alguna variable aún sin valores, ir a la antepenúltima variable y poner CUATRO valores V y luego CUATRO valores F alternadamente continuar hasta llegar a la primera variable (si hay alguna variable aún sin valores) siempre poniendo en la siguiente variable (de atrás hacia adelante) el doble de valores V que los del paso anterior, y luego similar cantidad de valores F alternadamente

Como se verá, al final se obtiene una tabla similar a la descrita antes. Lo más importante de esta etapa de la construcción de la tabla de verdad es que **no deben repetirse combinaciones** y los métodos anteriores le ayudan a construir la misma de un modo que evita este posible error.

Paso 1
Última variable (s)
(1V,1F,1V,1F...)

p	q	r	s	...
			V	...
			F	...
			V	...
			F	...
			V	...
			F	...
			V	...
			F	...
			V	...
			F	...
			V	...
			F	...
			V	...
			F	...
			V	...
			F	...
			V	...
			F	...

Paso 2
Penúltima (r)
(2V,2F,2V...)

p	q	r	s	...
		V	V	...
		V	F	...
		F	V	...
		F	F	...
		V	V	...
		V	F	...
		F	V	...
		F	F	...
		V	V	...
		V	F	...
		F	V	...
		F	F	...
		V	V	...
		V	F	...
		F	V	...
		F	F	...

Paso 3
Antepenúltima(q)
(4V,4F,...)

p	q	r	s	...
	V	V	V	...
	V	V	F	...
	V	F	V	...
	V	F	F	...
	F	V	V	...
	F	V	F	...
	F	F	V	...
	F	F	F	...
	V	V	V	...
	V	V	F	...
	V	F	V	...
	V	F	F	...
	F	V	V	...
	F	V	F	...
	F	F	V	...
	F	F	F	...

Paso 4
Primera variable (p)
(8V,8F,...)

p	q	r	s	...
V	V	V	V	...
V	V	V	F	...
V	V	F	V	...
V	V	F	F	...
V	F	V	V	...
V	F	V	F	...
V	F	F	V	...
V	F	F	F	...
F	V	V	V	...
F	V	V	F	...
F	V	F	V	...
F	V	F	F	...
F	F	V	V	...
F	F	V	F	...
F	F	F	V	...
F	F	F	F	...

Construcción de tablas de verdad para proposiciones compuestas

Después que se ha preparado la tabla de verdad según los pasos anteriores, lo que sigue es evaluar la proposición compuesta. Volvemos al ejemplo inicial visto en la sección anterior, o sea, construir la tabla de verdad de la proposición $p \wedge q \wedge r$ la tabla con sus 8 filas según lo visto en la sección anterior.

Para evaluar la primera fila tendríamos que: p = V, q = V, r = V lo que implica que se debe colocar el valor de $V \wedge V \wedge V$ lo cual resulta igual a V. Lo mismo debe hacerse para cada una de las filas o interpretaciones hasta completar la tabla quedando la misma como sigue.

p	q	r	$p \wedge q \wedge r$	Justificación para la interpretación
V	V	V	V	$V = V \wedge V \wedge V$
V	V	F	F	$F = V \wedge V \wedge F$
V	F	V	F	$F = V \wedge F \wedge V$
V	F	F	F	$F = V \wedge F \wedge F$
F	V	V	F	$F = F \wedge V \wedge V$
F	V	F	F	$F = F \wedge V \wedge F$
F	F	V	F	$F = F \wedge F \wedge V$
F	F	F	F	$F = F \wedge F \wedge F$

Como puede verse, sólo la primera fila es verdadera ya que es la única interpretación que tiene valores V para todas las proposiciones y por tanto es el único donde la conjunción es verdadera. En ocasiones, es conveniente añadir columnas para partes de la proposición compuesta y de esta forma obtener la tabla de verdad deseada paso a paso.

Por ejemplo, si la proposición fuera $(p \wedge q) \vee r$ la tabla de verdad quedaría así.

p	q	r	$p \wedge q$	$(p \wedge q) \vee r$
V	V	V	V	V
V	V	F	V	V
V	F	V	F	V
V	F	F	F	F
F	V	V	F	V
F	V	F	F	F
F	F	V	F	V
F	F	F	F	F

Para este caso y con la intención de hacer más fácil la elaboración de la tabla de verdad, es conveniente hacer una columna para $p \wedge q$ y hacer más fácil el cálculo final.

Como se había dicho antes, el uso de los paréntesis es imprescindible en algunos casos como es este. Veamos ahora la tabla de verdad de la otra proposición compuesta que vimos antes y que sólo de diferencia de esta en los paréntesis. Si la proposición fuera $p \wedge (q \vee r)$ la tabla de verdad quedaría así.

p	q	r	q ∨ r	p ∧ (q ∨ r)
V	V	V	V	V
V	V	F	V	V
V	F	V	V	V
V	F	F	F	F
F	V	V	V	F
F	V	F	V	F
F	F	V	V	F
F	F	F	F	F

Si se observan ambas tablas de verdad vemos que son diferentes pues hay interpretaciones que son diferentes (en este caso son dos y se han marcado en rojo). En este caso, hay dos interpretaciones que para (p ∧ q) ∨ r son modelos, mientras que para p ∧ (q ∨ r) son contraejemplos o contramodelos. Si vemos estos caso en detalles veremos que:

p=F, q=V, r=V
para (p ∧ q) ∨ r queda (F ∧ V) ∨ V = (F) ∨ V = F ∨ V = V (modelo)
para p ∧ (q ∨ r) queda F ∧ (V ∨ V) = F ∧ (V) = F ∧ V = F (contramodelo)

p=F, q=F, r=V
para (p ∧ q) ∨ r queda (F ∧ F) ∨ V = (F) ∨ V = F ∨ V = V (modelo)
para p ∧ (q ∨ r) queda F ∧ (F ∨ V) = F ∧ (V) = F ∧ V = F (contramodelo)

Como puede verse, de esta manera queda demostrada la necesidad de los paréntesis en este caso. Otro caso en que es muy útil escribir columnas intermedias antes de calcular la proposición compuesta que se desea en cuando hay proposiciones negadas. Por ejemplo, la tabla de verdad de p ∧ -q quedaría así.

p	q	-q	p ∧ -q
V	V	F	F
V	F	V	V
F	V	F	F
F	F	V	F

La columna –q nos ayuda a evaluar finalmente la proposición p ∧ -q.

Tautologías, contradicciones y contingencias.

Como ya hemos visto, en las tablas de verdad hay interpretaciones que son modelos ya que en ellos la proposición analizada toma valor V mientras que otras interpretaciones son contramodelos ya que en ellos la proposición analizada toma valor V.

Tautologías

Existen algunas proposiciones compuestas especiales en los que todas las interpretaciones son modelos, o sea, donde siempre el valor de la proposición analizada es V. A estas proposiciones se les llama tautologías.

Veamos algunos ejemplos de tautologías. El primero es $p \vee -p$.

p	-p	$p \vee -p$
V	F	V
F	V	V

En este caso tan simple la proposición es siempre verdadera ya que siempre tiene que cumplirse una proposición o su negado. Esto está intrínsecamente ligado a la definición que se hizo de qué es una proposición.

Otra tautología es $(p \wedge q) \vee (-p \wedge q) \vee -q$:

p	q	-p	-q	$(p \wedge q)$	$(-p \wedge q)$	$(p \wedge q) \vee (-p \wedge q) \vee -q$
V	V	F	F	**V**	F	V
V	F	F	**V**	F	F	V
F	V	V	F	F	**V**	V
F	F	V	**V**	F	F	V

Este caso es más complejo, pero podemos ver nuevamente en la tabla de verdad que todas las interpretaciones son modelos y que por tanto la proposición $(p \wedge q) \vee (-p \wedge q) \vee -q$ también es una tautología.
Las tautologías también se dice que son proposiciones o fórmulas "semánticamente válidas" ya que no tienen contraejemplos. Y normalmente para representarlo se usa el símbolo de fórmula válida " ⊢ ". Las anteriores serían:
⊢ $p \vee -p$
⊢ $(p \wedge q) \vee (-p \wedge q) \vee -q$

Contradicciones

Existen algunas proposiciones compuestas especiales en los que todas las interpretaciones son contramodelos o contraejemplos, o sea, donde siempre el valor de la proposición analizada es F. A estas proposiciones se les llama contradicciones pues no hay ninguna interpretación que las satisface o le hace valor "verdadero" (V).

Veamos algunos ejemplos de contradicciones. El primero es $p \wedge -p$.

p	-p	$p \wedge -p$
V	F	F
F	V	F

En este caso tan simple la proposición es siempre falsa ya que nunca puede cumplirse una proposición y su negado a la vez, por tanto es una contradicción. Esto también tiene que ver directamente con la definición de proposición.

Otra contradicción es $(p \vee q) \wedge (-p \vee q) \wedge -q$:

p	q	-p	-q	$(p \vee q)$	$(-p \vee q)$	$(p \vee q) \wedge (-p \vee q) \wedge -q$
V	V	F	**F**	V	V	F
V	F	F	V	V	**F**	F
F	V	V	**F**	V	V	F
F	F	V	V	**F**	V	F

Este caso es más complejo, pero podemos ver nuevamente en la tabla de verdad que todas las interpretaciones son contramodelos y por tanto la proposición $(p \vee q) \wedge (-p \vee q) \wedge -q$ también es una contradicción.

Contingencias

Al resto de las proposiciones que tiene tanto modelos como contraejemplos reciben el nombre de contingencias. La mayor parte de las proposiciones que hemos visto son contingencias. Los casos más simples son las definiciones dadas para las conectivas negación, conjunción y disyunción. En sus tablas de verdad se puede ver que hay modelos y contramodelos. Las contingencias y las tautologías reciben también el nombre de fórmulas o proposiciones "satisfacibles" ya que algunas de sus interpretaciones son modelos (para el caso de las tautologías todas sus interpretaciones lo son)

Otros casos de proposiciones que son contingencias son:

p	q	-p	$(p \vee q)$	$(-p \vee q)$	$(p \vee q) \wedge (-p \vee q)$
V	V	F	V	V	V
V	F	F	V	F	F
F	V	V	V	V	V
F	F	V	F	V	F

p	q	-p	$(p \wedge q)$	$(-p \wedge q)$	$(p \wedge q) \vee (-p \wedge q)$
V	V	F	V	F	V
V	F	F	F	F	F
F	V	V	F	V	V
F	F	V	F	F	F

p	q	-p	-q	$(p \wedge q)$	$(-p \wedge -q)$	$(p \wedge q) \vee (-p \wedge -q)$
V	V	F	F	V	F	V
V	F	F	V	F	F	F
F	V	V	F	F	F	F
F	F	V	V	F	V	V

p	q	-p	-q	$(p \vee q)$	$(-p \vee -q)$	$(p \vee q) \wedge (-p \vee -q)$
V	V	F	F	V	F	F
V	F	F	V	V	V	V
F	V	V	F	V	V	V
F	F	V	V	F	V	F

Tema II: Proposiciones condiciones y bicondicionales. Equivalencia lógica.

Proposiciones condicionales

Frases en español que representan proposiciones condicionales

En el lenguaje de la ciencia, de las leyes y en la vida normal son muy comunes afirmaciones condicionales. En este tipo de frases no se afirma directamente que algo se cumple como ocurre con las frases vistas anteriormente. Las afirmaciones condicionales afirman que una conclusión determinada es válida cuando se cumpla una condición que se exige. Por tanto, al analizar una proposición condicional es fundamental determinar cual es la condición y cual es la conclusión.

La frase básica para denotar una proposición condicional es: "si **p** entonces **q**" siendo **p** y **q** proposiciones. La proposición que ocupa el lugar de **p** recibe el nombre de condición, mientras que la que ocupa el lugar de **q** recibe el nombre de conclusión.

Hay muchas otras maneras de expresar una relación similar, lo único que debe tenerse claro para reconocerlo es que la frase debe afirmar que en todos los casos en que se cumpla **p** debe cumplirse también **q** (o sea que el cumplimiento de la condición **p** es suficiente para afirmar que la conclusión **q** tiene que cumplirse también). Dicho de otra manera, si **q** no cumplió entonces no es posible que **p** se haya cumplido (o sea que el cumplimiento de la conclusión **q** es necesaria para afirmar que condición **p** se cumpla también). Es importante notar que entre **p** y **q** no se puede hablar de una causa o un efecto, de ninguna manera se está afirmando una relación de este tipo solo se está determinando una relación entre los valores de verdad de ambas proposiciones. Saber la causa de cada hecho es algo más complicado. Por tanto para determinar en una frase condicional cual es la condición **p** debe basarse en la identificación de la proposición que es suficiente para afirmar la conclusión **q**. Por otra parte, para determinar en una frase condicional cual es la conclusión **q** debe basarse en la identificación de la proposición que es necesaria para afirmar la condición **p**.

Resumiendo, la condición **p** es lo suficiente mientras que la conclusión **q** es lo necesario.

Hay frases que hacen más énfasis en la condición y otras en la conclusión. Entre las frases en Español que denotan una proposición condicional $\mathbf{p}\rightarrow\mathbf{q}$ están:

- "***si llueve entonces me mojo***" donde se está afirmando que siempre que llueva me voy a mojar, por eso se dice que la proposición "llueve" es suficiente para hacer cierta la proposición "me mojo" por esta razón la condición p="llueve" y la conclusión q="me mojo". El esquema de esta frase "***si p entonces q***".
- "***si llueve, me mojo***" esta frase es similar a la anterior, pero la coma hace la función de "entonces", de manera similar al caso anterior se identifica la condición p="llueve" y la conclusión q="me mojo" El esquema de la frase es "***si p, q***".
- "***me mojo si llueve***", este ejemplo vuelve a ser similar y solo se han movido de lugar las proposiciones pero sigue siendo suficiente la condición p="llueve" para afirmar la conclusión q="me mojo" . El esquema de esta frase es "***q si p***".
- "***siempre que llueve me mojo***", en este caso se está dejando claro que no hay posibilidades de que llueva y no me moje, por eso se aprecia que basta o es suficiente la condición p="llueve" siendo el esquema de esta frase "***siempre que p [entonces] q***" (en el esquema puede obviarse el uso de "entonces")
- "***llueve solo si me mojo***" o lo que es lo mismo "***ha llovido solo si me he mojado***", aquí se está afirmando de manera explícita que es necesaria la proposición q="me mojo" para afirmar la condición p="llueve" pues se está excluyendo la posibilidad

de que "llueva" y yo no "me moje". El esquema de esta frase es "***p sólo si q***". Otro ejemplo de frase con este esquema es la que en ocasiones se dice antes de una prueba muy difícil que no se puede aprobar sin estudiar o sea que "***si aprueba entonces estudió***"... muchas veces los profesores antes de estas pruebas dicen "***aprobarán solo si estudian***".

- "***solo si me mojo, llueve***", el análisis de esta frase es similar a la anterior pero sólo se ha cambiado de orden las proposiciones y por esto se ha agregado la coma. Se está afirmando de manera explícita que es necesaria la proposición q="me mojo" para afirmar la condición p="llueve". El esquema de esta frase es "***solo si q, p***"
- "***cuando llueve ocurre que me mojo***", aquí el "cuando" está fijando una condición que es suficiente (la condición p="llueve") para que cumpla la proposición q="me mojo" El esquema de esta frase es "***cuando p ocurre q***" (también puede sustituirse ocurre por entonces o por una coma en el esquema)
- "***me mojo cuando llueve***", esta es similar a la anterior sólo variando el orden pero se está fijando una condición que es suficiente (la condición p="llueve") para que cumpla la proposición q="me mojo" El esquema de esta frase es "***q cuando p***".
- "***una condición necesaria para que llueva es que me moje***", esta frase sigue la línea de las anteriores haciendo explícita la necesidad de la proposición q="me mojo" para se cumpla la proposición p="llueve". El esquema de esta frase es "***una condición necesaria para p es q***". Estas frases son fáciles de interpretar porque se está diciendo de manera explícita lo que necesario y por tanto quién es **q**.
- "***es necesario que me moje para que llueva*** " o si se quiere "***es necesario que me moje para que haya llovido ,*** es similar al ejemplo anterior solo variando el orden de las proposiciones. El esquema de la frase es "***es necesario q para p***".
- "***que me moje es necesario para que llueva***, otra variación de lo anterior con el esquema de frase "***q es necesaria para p***"
- "***una condición suficiente para que me moje es que llueva***", aquí el esquema es "***una condición suficiente para q es p***". Estas frases son fáciles de interpretar porque se está diciendo de manera explícita lo que suficiente y por tanto quién es **p**.
- "***que llueva es suficiente para que me moje***", aquí el esquema es "***p es suficiente para q***"
- "***es suficiente que llueva para que me moje***", aquí el esquema es "***es suficiente p para q***"
- "***basta que llueva para que me moje***", aquí se deja claro que es suficiente (basta) "que llueva" y el esquema es "***basta q para p***"
- "***no llueve a menos que me moje***", aquí se vuelve a dejar clara la necesidad de q y el esquema es "***no p a menos que q***".

Puede ser muy aburrida esta enumeración de frases. Sin embargo debe quedar claro que existen otros mucho esquemas de frases para expresar proposiciones condicionales,

Veamos algunas variantes para la frase: "si un cuerpo está libre entonces la tierra lo atrae"

- "si un cuerpo está libre entonces la tierra lo atrae".
- "siempre que un cuerpo está libre la tierra lo atrae"
- "un cuerpo está libre sólo si la tierra lo atrae"

- "cuando un cuerpo está libre la tierra lo atrae
- "una condición necesaria para que un cuerpo esté libre es que la tierra lo atraiga".
- "una condición suficiente para la tierra atraiga un cuerpo es que esté libre"
- "un cuerpo no está libre a menos que la tierra lo atraiga".

Y también puede haber otras maneras diferentes a nuestro largo listado como "la tierra atrae a los cuerpo que están libres", "ningún cuerpo libre deja de ser atraído por la tierra", etc.
Hay frases en que queda aún menos claramente dicha la expresión condicional como puede ser en la poesía y en general depende del estilo del que habla. Por ejemplo, en la famosa frase en el juicio del Moncada "Condenadme, no importa, la historia me absolverá" hay una expresión condicional. Sin perder nada de su sentido original la frase es equivalente a decir "no importa que me condenen porque la historia me absolverá" o "no me importa que me condenen si la historia me absolverá" que ya pertenece al esquema de frase "q si p" ya que se puede decir que la condición p="la historia me absolverá" es suficiente para afirmar la conclusión q="condenadme, no importa". En esta transformación de la frase de Fidel se está dejando explícitamente qué es causa y qué es consecuencia usando de manera implícita la expresión "porque" representada en su frase original por la segunda coma. Como puede verse, en ocasiones cuesta un poco de esfuerzo reconocer correctamente las expresiones condicionales. En ocasiones se usan frase como "debido a", "por tanto", "a causa de", etc.

Tablas de verdad para la conectiva condicional o implicación

A la hora de construir la tabla de verdad de la proposición condicional **p→q** debemos tener muy en cuanta el significado de la misma. Como se ha dicho, la proposición condicional **p→q** debemos entenderla como "si p se cumple entonces q tiene también que cumplirse" o también como "solo puede cumplirse p si se cumple q". La diferencia entre una y otra es sólo un problema del sentido de lo que se quiere reforzar, pero lo que sí debe quedar claro es que en la interpretación en que p=V debe cumplirse q=V para que la proposición condicional **p→q** sea cierta. De aquí entonces podemos entender claramente que los valores de verdad para **p→q** cuando p=V quedan como se muestra en la tabla de verdad siguiente.

p	q	**p→q**
V	V	**V**
V	F	**F**
F	V	**V**
F	F	**V**

La razón es muy simple. Si p se cumple (p=V) y q no se cumple (q=F) entonces no se puede decir que **p→q** porque realmente p no esta implicando q ya que ocurrió el primero pero no ocurrió el segundo. Por otra parte, la primera fila de la tabla es más clara aún ya que se cumplió la condición p y se cumplió también q, lo cual hace que la proposición condicional **p→q** sea válida.

Sin embargo, en un sentido puramente semántico no queda claro qué debe pasar cuando p no se cumpla que son los casos marcados en rojo. En estos casos la proposición condicional **p→q** no tiene sentido aplicarla. Por ejemplo, si se cumpliera el siguiente significado para las proposiciones p y q:

p : Estoy contento
q : Voy al cine

La proposición condicional **p→q** significa "si estoy contento entonces voy al cine". Si usted está contento entonces el grado de verdad de la proposición condicional **p→q** queda dependiendo de si finalmente voy al cine o no. Si finalmente va al cine entonces se cumple lo que dijo. Por otra parte, si no va al cine a pesar de estar contento entonces no se cumple lo que dijo y por tanto la frase condicional tiene valor falso, lo que sería similar a decir que dijo una mentira. Ahora, si no está contento entonces la proposición condicional **p→q** no es aplicable ya que lo que se dijo no se refería a este caso, entonces vaya o no vaya al cine no se puede decir que la frase condicional que se analiza sea verdadera o falsa. Es decir, que si la condición no se satisface, entonces nadie puede afirmar que se dijo mentira independiente de lo que se haga, como tampoco se puede afirmar que fuera verdad. **Sin embargo, por convenio se asume que el valor de verdad de la proposición condicional p→q en el caso que la condición no se cumple es V como se mostró en rojo en la tabla de verdad anterior.** Asumir esto es equivalente a pensar que la frase condicional dicha es verdadera (V) tanto en los casos donde se cumple su sentido (p=V, q=V) como en los casos donde no es aplicable (p=F para cualquier valor de q). Por tanto, la única interpretación que es contramodelo o contraejemplo de la implicación o conectiva condicional sería cuando (p=V, q=F) ya que se cumplió la condición exigida y no se cumplió la conclusión prevista. Resumiendo, la tabla de verdad queda como sigue.

p	q	**p→q**
V	V	**V**
V	F	**F**
F	V	**V**
F	F	**V**

En algunos textos puede encontrarse que esta conectiva se le da el nombre de "implicación material", aunque es más frecuente que se le llame implicación.

Recíproca de una proposición condicional

La recíproca de una proposición condicional **p→q** es la proposición condicional **q→p**. Aunque aparentemente parezcan muy similares realmente no son lo mismo. Para el mismo significado que se vio en la sección anterior, cada una se interpreta de manera distinta:

p→q significa **"**si estoy contento entonces voy al cine**"**
p→q significa **"**si voy al cine entonces estoy contento**"**

Es evidente que no dicen lo mismo ya que para la primera proposición condicional si usted no está contento y a pesar de eso fue al cine (es decir p=F, q=V) no provoca un valor falso

en la proposición **p→q** ya que la regla no es aplicable en ese caso porque p=F y por tanto es verdadera. Sin embargo, para la proposición **q→p** los mismo valores de verdad correspondientes a que usted no está contento y a pesar de eso fue al cine (es decir p=F, q=V) sí provoca un valor falso a la proposición **q→p** ya que se cumplió su antecedente q y no se cumplió su conclusión p. Igualmente provoca son diferentes los valores de verdad de ambas proposiciones si usted no fue al cine a pesar de estar contento (es decir p=V, q=F). En este caso, la proposición **p→q** sería falsa, mientras que la proposición **q→p** va a ser verdadera.
Si hacemos una tabla de verdad donde pongamos juntas ambas proposiciones podemos observar la diferencia. Se han marcado en rojo las dos interpretaciones que son diferentes.

p	q	**p→q**	**Ya que ...**	**q→p**	**Ya que ...**
V	V	V	**V→V=V**	V	**V→V=V**
V	F	**F**	V→F=F	**V**	F→V=V
F	V	**V**	F→V=V	**F**	V→F=F
F	F	**V**	F→F=V	**V**	F→F=F

A partir de esta tabla de verdad podemos dejar sentado que una proposición y su recíproca no son equivalentes en el sentido que no toman siempre los mismos valores de verdad. Sobre el concepto de equivalencia se volverá más adelante.

Contrapositiva de una proposición condicional

La contrapositiva (también llamada contraposición o trasposición) de una proposición condicional **p→q** es la proposición condicional **-q→ -p**. Aunque aparentemente parezcan diferentes similares realmente son equivalentes. Para el mismo significado que se vio en la sección anterior, la interpretación de cada una es la siguiente:

p→q significa **"**si estoy contento entonces voy al cine**"**
-q→ -p significa **"**si no voy al cine entonces no estoy contento**"**

Es evidente que dicen lo mismo lo que una afirma lo que se cumple haciendo énfasis en que si la condición p se cumple entonces se tiene que cumplir la condición q; mientras que su contrapositiva hace el énfasis en que si la proposición p no se cumplió entonces la proposición tampoco pudo cumplirse. No existen interpretaciones diferentes para ambas proposiciones. Si hacemos una tabla de verdad donde pongamos juntas ambas proposiciones podemos observarlo.

p	q	**p→q**	**Ya que ...**	**-p**	**-q**	**- q→ - p**	**Ya que ...**
V	V	V	**V→V=V**	**F**	**F**	V	**F→F=V**
V	F	**F**	V→F=F	F	V	**F**	V→F=F
F	V	**V**	F→V=V	V	F	**V**	F→V=V
F	F	**V**	F→F=V	V	V	**V**	V→V=V

A partir de esta tabla de verdad podemos dejar sentado que una proposición y su contrapositiva son equivalentes en el sentido que toman siempre los mismos valores de verdad.

Proposiciones bicondicionales

Frases en español que representan proposiciones bicondicionales

En cuanto a las frases en inglés que denotan expresiones bicondicionales, afortunadamente la situación es mucho más simple que para las condicionales simples. Entre las frases más conocidas están:

- ”p es necesario y suficiente para q”, por ejemplo “que un numero sea divisible por dos es necesario y suficiente para decir que es par”
- “p si y solo si q”, por ejemplo “un número es primo si y solo si solamente es divisible por 1 y por sí mismo”
- p es equivalente a decir q
- si p entonces q y viceversa

También son válidas todas las otras frases que se vieron para las condicionales si se le agrega además las frases “y viceversa” para indicar que ambas proposiciones están condicionadas en ambos sentidos. O sea, que p es condición suficiente y necesaria para q, lo cual equivale también a decir que q es condición necesaria y suficiente para p.

El hecho de que la relación sea en los dos sentidos hace más fácil el reconocer correctamente las frases bicondicionales.

Tablas de verdad para la conectiva bicondicional o doble implicación.

La proposición bicondicional **p↔q** en su sentido semántico no deja espacios a ninguna duda. Como se ha dicho esta proposición significa que p debe cumplirse SI y SOLO SI se cumple q. Por tanto, no hay posibilidades de que una de las proposiciones sea verdadera (V) y la otra falsa (F). Por esta razón, sus interpretaciones donde ambas proposiciones sean verdaderas es el primer caso que queda clara su semántica que debe ser verdadera la proposición bicondicional **p↔q.** Queda igualmente claro que si una de las dos es verdadera y la otra falsa entonces la proposición bicondicional **p↔q** debe tomar valor falso ya que no se cumplió la implicación condicional en uno de los sentidos. Por último, si ambas son falsas, la proposición bicondicional **p↔q** debe ser también verdadera ya que ella significa que una de ellas puede cumplirse SI y SOLO SI se cumple la otra, por tanto si ninguna de las dos se cumple se esta cumpliendo también este criterio. Por esta razón, y resumiendo, el valor de la proposición bicondicional **p↔q** debe ser verdadero (V) si ambas proposiciones valen lo mismo (ya sea verdadero o falso) y debe ser falsa si son diferentes. La tabla de la verdad de esta conectiva queda como sigue.

p	q	**p↔q**
V	V	**V**
V	F	**F**
F	V	**F**
F	F	**V**

En algunos textos puede encontrarse que esta conectiva se le da el nombre de "equivalencia material", aunque es más frecuente que se le llame doble implicación.

Relación entre las proposiciones condicionales y bicondicionales.

Las proposiciones bicondicionales reciben este nombre ya que se dice que están compuestas por dos proposiciones condicionales. O sea, **p↔q** significa que **p→q** y que **q→p.** O sea, que ninguna de las dos es condición ni conclusión, o si se quiere que ambas lo son. Por esta razón, en el caso de las proposiciones bicondicionales es incorrecto usar los términos condición y conclusión.

Precisando un poco más lo dicho, la proposición condicional **p↔q** es equivalente a la conjunción de una condicional y su recíproca, o sea a la proposición **(p→q)∧(q→p).** Como hemos visto también antes, una manera de verificar esto es construyendo la tabla de verdad de ambas y comprobando que para cada interpretación los valores de ambas proposiciones son iguales. Esto se comprueba en la tabla de verdad siguiente.

p	q	**p↔q**	**(p→q)**	**(q→p)**	**(p→q)∧(q→p)**
V	V	**V**	V	V	**V**
V	F	**F**	**F**	**V**	**F**
F	V	**F**	**V**	**F**	**F**
F	F	**V**	**V**	**V**	**V**

Equivalencias lógicas

Concepto de proposiciones lógicamente equivalentes

Desde el primer tema se viene hablando de proposiciones que son equivalentes. En este momento vamos a finalmente precisar este concepto.

Desde el punto de vista semántico, se dice que dos proposiciones son lógicamente equivalentes si ambas tienen los mismos valores de verdad para todas las combinaciones de valores de las proposiciones simples que las componen. Es decir, en cada una de las interpretaciones de ambas, los valores de verdad de ambas proposiciones son iguales. Puede por tanto decirse que todos los modelos de una son también modelos de la otra y viceversa. Esto es lo mismo que decir que todos los modelos de una son modelos de la otra, y todos los contramodelos de una también lo son de la otra. En el resto del curso se usará la palabra "equivalente" como sinónimo de "lógicamente equivalente".

Desde el primer tema hemos visto que las tablas de verdad son una herramienta semántica de gran utilidad para verificar si dos proposiciones son equivalentes o no. Para hacerlo, se construye una tabla de verdad y se calculan los valores de verdad para cada columna que se corresponde con cada proposición. Una vez hecho esto, se debe comprobar que ambas columnas siempre tienen el mismo valor. Si ocurre esto, entonces ambas son equivalentes. Si hay alguna diferencia entonces no lo son.

Una manera alternativa para chequear si dos proposiciones son equivalentes es verificando si la proposición bicondicional que resulta de conectar a ambas con la conectiva bicondicional es una tautología. Es decir dos proposiciones v y w son equivalentes si la proposición compuesta **v↔w** es una tautología.

La equivalencia entre dos proposiciones se representa usando el símbolo de equivalencia (≡) que es similar al signo de igualdad (=) pero con tres rayas. Por su parte la no equivalencia se representa con el símbolo de no equivalencia (≠).

Veamos un ejemplo de dos proposiciones que sabemos que son equivalentes: la proposición condicional **p→q** y su contrapositiva **-q→ -p**. Para verificar que son equivalentes se puede hacer la tabla de verdad de ambas y ver si sus interpretaciones tienen iguales valores de verdad, o podemos verificar que la proposición **(p→q)↔(-q→ -p)** es una tautología. Esto último también se puede verificar en una tabla de verdad. Veámosla.

p	q	**p→q**	**-p**	**-q**	**- q→ - p**	**(p→q)↔(-q→ -p)**
V	V	**V**	F	F	**V**	**V**
V	F	**F**	F	V	**F**	**V**
F	V	**V**	V	F	**V**	**V**
F	F	**V**	V	V	**V**	**V**

Puede apreciarse que por cualquier de los dos métodos llegamos a la conclusión de que ambas proposiciones son equivalentes. Por una parte, para todas las interpretaciones los valores de verdad de ambas es el mismo. Por la otra vía se puede ver que la proposición bicondicional conectando a ambas es una tautología.
Veamos ahora un ejemplo de dos proposiciones que sabemos que no son equivalentes: la proposición condicional **p→q** y su recíproca **q→ p**. En este caso la proposición que debemos comprobar si es o no tautología es **(p→q)↔(q→p)**. La tabla de verdad queda así.

p	q	**p→q**	**q→p**	**(p→q)↔(q→p)**
V	V	V	V	V
V	F	**F**	**V**	**F**
F	V	**V**	**F**	**F**
F	F	V	V	V

Puede apreciarse que por cualquier de los dos métodos llegamos a la conclusión de que ambas proposiciones no son equivalentes. Por una parte, hay dos interpretaciones (marcadas en rojo) para las que los valores de verdad de ambas son diferentes. Por la otra vía se puede ver que la proposición bicondicional conectando a ambas no es una tautología sino una contingencia ya que tiene dos modelos y dos contramodelos o contraejemplos. Por tanto **p→q** y **q→p** no son equivalentes.
Como se dijo, la equivalencia entre dos proposiciones se representa usando el símbolo de equivalencia (≡) y la no equivalencia se representa con el símbolo de no equivalencia (≠). Por tanto, la equivalencia entre una proposición y su contrapositiva se representa:

p→q ≡ -q → -p

Por otra parte, la no - equivalencia entre una proposición y su recíproca se representa:

p→q ≠ q→p

Por último veamos un ejemplo muy popular dentro de la lógica de proposiciones equivalentes que son las llamadas **leyes de De Morgan**. El nombre de estas leyes se debe a un célebre matemático inglés llamado Augustus De Morgan (27/Junio/1806,India - 18 Marzo 1871, Inglaterra). Las leyes de De Morgan son las dos siguientes:

- $-(p \vee q) \equiv -p \wedge -q$
- $-(p \wedge q) \equiv -p \vee -q$

Para demostrar la primera de estas equivalencias veremos nuevamente la tabla de verdad de las proposiciones analizadas para comprobar si los valores de verdad de **-(p∨q)** y de **-p∧-q** son siempre iguales en todas las interpretaciones o alternativamente que si la proposición **-(p∨q) ↔ -p ∧ -q** es una tautología. La tabla queda como sigue y en ella puede comprobarse que **-(p∨q) ≡ -p ∧ -q**

p	q	**p∨q**	**-(p∨q)**	**-p**	**-q**	**-p ∧ -q**	**-(p∨q) ↔ -p ∧ -q**
V	V	**V**	F	**F**	**F**	F	V
V	F	V	**F**	F	V	**F**	**V**
F	V	V	**F**	V	F	**F**	**V**
F	F	F	**V**	V	V	**V**	**V**

Para comprobar la otra ley de De Morgan volveremos a construir la tabla de verdad de ambas proposiciones y en ella volvemos a comprobar la equivalencia, en este caso entre **-(p∧q)** y **-p ∨ -q.**

p	q	**p∧q**	-(p∧q)	**-p**	**-q**	-p ∨ -q	**-(p∧q) ↔ -p ∨ -q**
V	V	**V**	F	F	F	F	V
V	F	F	V	F	V	V	**V**
F	V	F	V	V	F	V	**V**
F	F	F	V	V	V	**V**	**V**

Proposiciones equivalentes a las proposiciones condicionales sin la conectiva condicional.

Existen proposiciones compuestas que son equivalentes a la proposición condicional a pesar de que no usan la conectiva condicional (→). Una de ellas es la que define la equivalencia siguiente: **p→q ≡ -p∨q**, la cual queda demostrada en la siguiente tabla.

p	q	p→q	**-p**	**-p∨q**	p→q ↔ -p∨q
V	V	V	F	V	V
V	F	F	F	F	V
F	V	V	V	V	V
F	F	V	V	V	V

Esta equivalencia es muy interesante para entender la semántica de la conectiva condicional o implicación.

La formula o proposición que se acaba de ver nos está dejando claro que la implicación es cierta o bien porque se cumple la conclusión **q** o cuando no se cumple la condición **p**. Esto deja muy claro que el único contraejemplo de la conectiva condicional es la interpretación en que se cumplió la condición **p** y no se cumplió la conclusión **q**, o sea la interpretación (p=V, q=F) correspondiente a la segunda columna de la tabla.

Otra equivalencia interesante es entre la negación de la proposición condicional. Esta equivalencia es **–(p→q) ≡ p∧-q** que aparece en el libro de texto en la página 14. Para demostrarla volvemos a hacer la tabla de verdad. Aquí la interpretación es que si es falso que **p→q** entonces es cierto que se cumplió **p∧-q.** (ver pág. siguiente)

p	q	p→q	-(p→q)	-q	p∧-q	-(p→q) ↔ p∧-q
V	V	V	F	F	F	V
V	F	F	V	V	V	V
F	V	V	F	F	F	V
F	F	V	F	V	F	V

Proposiciones equivalentes a las proposiciones bicondicionales sin la conectiva bicondicional.

En la sección 0 se mostró la equivalencia **p↔q ≡ (p→q)∧(q→p).** Aquí y sólo con el objetivo de completar la tabla de verdad haciendo la comprobación de que la proposición bicondicional **(p↔q) ↔ ((p→q)∧(q→p))** es una tautología se repite aquí la tabla de verdad con una columna más. En este caso los paréntesis complican un poco la lectura pero dejan sin ambigüedades el significado.

p	q	p↔q	(p→q)	(q→p)	(p→q)∧(q→p)	(p↔q) ↔ ((p→q)∧(q→p))
V	V	**V**	V	V	**V**	**V**
V	F	**F**	F	V	**F**	**V**
F	V	**F**	V	F	**F**	**V**
F	F	**V**	V	V	**V**	**V**

Sin embargo, hay otras proposiciones que son equivalentes, una de ellas es la siguiente equivalencia **p↔q ≡ (p∧q)∨(-p∧-q).** Si analizamos su semántica nos damos cuanta que significa que la doble implicación o conectiva bicondicional se cumple si ambas proposiciones son verdaderas o si ambas son falsa, o sea, si son iguales. Comprobando en la tabla de verdad vemos que es cierta la equivalencia.

p	q	p↔q	-p	-q	p∧q	-p∧-q	(p∧q)∨(-p∧-q)	(p↔q) ↔ ((p∧q)∨(-p∧-q))
V	V	**V**	F	F	V	F	**V**	V
V	F	**F**	F	V	F	F	**F**	**V**
F	V	**F**	V	F	F	F	**F**	**V**
F	F	**V**	V	V	F	V	**V**	**V**

Si se quiere comprobar la equivalencia **p↔q ≡ (p∨-q)∧(-p∨q).** En este caso el sentido vuelve a ser claro ya que se está exigiendo que los valores sean iguales, porque si no en uno de los términos que está dentro de los paréntesis va a ser falso y por tanto la conjunción de ese con el otro término también lo será.

p	q	p↔q	-p	-q	p∨-q	-p∨q	(p∨-q)∧(-p∨q)	(p↔q) ↔ ((p∨-q)∧(-p∨q))
V	V	**V**	F	F	V	V	**V**	**V**
V	F	**F**	F	V	V	F	**F**	**V**
F	V	**F**	V	F	F	V	**F**	**V**
F	F	**V**	V	V	V	V	**V**	**V**

Tema III: Algoritmización. Introducción a la algoritmización

Como ya se ha dicho, las computadoras electrónicas digitales constituyen potentes herramientas que ayudan al hombre a resolver problemas de diversa naturaleza. En aras de lograr este objetivo, las computadoras deben ser programadas para que puedan desarrollar la solución a dichos problemas en función de una colección de condiciones presentes en el momento de dar esta solución.
Hasta ahora, se han estudiado elementos de la lógica, importantes para la programación de computadoras electrónicas digitales.
El término de algoritmización se refiere a la manera de representar los pasos para llegar a la solución de los problemas que se quieren resolver con la ayuda de la computadora. Esta solución se expresa en un lenguaje o notación más o menos estándar que permita su traducción a los lenguajes de programación, o sea, los lenguajes con los que se programan las computadoras.
La palabra algoritmo proviene del nombre del matemático árabe al-Khowârizmî, quien definió diversas secuencias de pasos en la solución de problemas algebraicos.

➢ Definición de algoritmo. Características

- **Definición de algoritmo:**

Un *algoritmo* es una secuencia lógica, finita y bien definida de pasos para dar solución a un tipo determinado de problemas.
Para una mejor comprensión de esta definición, se analizará por partes.
En primer lugar, el algoritmo es una secuencia o conjunto de pasos que hay que desarrollar para llegar a la solución del problema. Esa secuencia tiene que ser lógica, o sea, debe especificar los pasos en un orden lógico, que guíe hacia la solución. Debe ser, además, finita y bien definida o precisa, como se explicará más adelante.

- **Características de los algoritmos:**

Para que una secuencia o conjunto de pasos cualquiera pueda ser considerado un algoritmo, debe tener las siguientes características:

- *Carácter finito:*

Un algoritmo debe siempre terminar luego de un número finito de pasos. Esta característica es muy importante, dado que la computadora debe dar la solución al problema planteado. Si el algoritmo no fuera finito puede suceder que nunca se alcance a la solución.

- *Precisión:*

Cada paso del algoritmo debe ser definido de manera precisa, rigurosa y sin ambigüedades.

Cuando los pasos del algoritmo se especifican de manera precisa y rigurosa, se entiende exactamente el significado de ellos. Cuando se especifican sin ambigüedades, sus resultados intermedios quedan definidos de manera única y sólo dependen de los datos de entrada y de los resultados de los pasos anteriores. Es decir, que la misma secuencia de pasos con los mismos datos de entrada no dan lugar a más de un resultado.

- *Entrada:*

Antes de comenzar su ejecución, el algoritmo recibe una entrada, o sea, se le proporciona información que será procesada por él. Cada elemento de la entrada se corresponde con un objeto del dominio del problema.

- *Salida:*

El algoritmo produce una salida, o sea, genera los resultados del procesamiento, los cuales guardan una relación específica con la entrada.

- *Efectividad:*

Cuando el algoritmo es efectivo, todas las operaciones que él realiza son suficientemente básicas como para que, en principio, puedan ser desarrolladas exactamente y en un tiempo finito por una persona, usando papel y lápiz.

- **Pasos para resolver un problema**

Para la solución de un problema, se ha de tener en cuenta la siguiente metodología general de trabajo:

1. Comprensión y análisis del problema planteado
2. Desarrollo de un modelo matemático que dé solución al problema
3. Desarrollo de la solución del problema mediante un algoritmo
4. Programación de la solución para la computadora

Es evidente que esta metodología constituye un algoritmo, que establece los pasos necesarios para dar solución a un problema.

En el primer paso de este algoritmo se estipula que el problema ha de entenderse correctamente antes de acometer su solución. Es obvio que no podremos dar solución a un problema sin entenderlo previamente, es decir, sin saber **qué** es lo que hay que resolver. Luego de entender el problema planteado, se analiza cuáles son sus características, en aras de enfocar correctamente su solución, o sea, **cómo** acometerla.

El segundo paso plantea que una vez que se ha comprendido y analizado el problema, se procede al desarrollo de un modelo matemático que presente la solución teórica del problema. Este modelo matemático, por supuesto, debe corresponderse con las características analizadas en el paso anterior, puede apoyarse con gráficos y fórmulas, según sea el caso. Es aquí donde debe quedar claro cuáles son los datos que son necesarios para el cálculo, o sea, los datos de entrada, cómo ellos son procesados, y cuáles son los resultados esperados de dicha solución, o sea la salida.

En el tercer paso se establece que con el modelo matemático están dadas las condiciones para desarrollar la secuencia de pasos –o sea, el algoritmo– necesaria para dar solución al problema.

El último de los pasos plantea el desarrollo, a partir del algoritmo, de un programa escrito en un lenguaje para la computadora, con el cual se le "instruye" de cuáles pasos seguir para resolver el problema.

➢ Precondiciones y poscondiciones

Siempre será necesario un buen estilo de solución de problemas, de modo tal que se obtengan **soluciones generales**, que haya **elevada productividad** en la programación, que se posibilite el **fácil mantenimiento** del código que se programe, que se desarrollen **soluciones suficientemente claras**, entre otros. Todos estos elementos se irán introduciendo en la medida en que se avance en el estudio de los algoritmos y de las estructuras que los componen.

Una de las cuestiones importantes de un buen estilo de solución de problemas tendrá en cuenta, para cada paso del algoritmo o para el algoritmo completo, las siguientes preguntas:
- ¿Qué condiciones deben cumplirse para poder llevar a cabo dicho paso?, y
- ¿Qué condiciones se cumplen luego de haber realizado el paso?

- **Precondiciones**

A las condiciones que dan respuesta a la primera de estas dos preguntas se les denomina *precondiciones* y tienen que cumplirse para poder desarrollar el paso correspondiente.

Ante cada paso de un algoritmo, se debe indagar por las precondiciones que éste pudiera tener asociadas. Esto asegura que no se realice el paso si no están dadas esas condiciones. Esto, sin dudas, garantiza que no se cometan errores y que se alcancen los resultados esperados o los correctos. En el momento en que se desarrolla el programa para la computadora, tener en cuenta las precondiciones pudiera permitir, además, tomar las medidas pertinentes para que el programa, incluso, no le permita al usuario suministrar datos incorrectos o en un orden inadecuado, por citar sólo dos ejemplos.

Ejemplo de precondición:

Un ejemplo de precondición está presente, sin dudas, cuando en un paso de un algoritmo se va a calcular la raíz cuadrada de un número. La precondición será, en ese caso, que el número sea no negativo.

- **Poscondiciones**

A las condiciones que dan respuesta a la segunda de las preguntas anteriores se les denomina *poscondiciones* y se cumplen siempre que se haya desarrollado el paso que las tiene asociadas o el algoritmo. Luego, las poscondiciones denotan el estado que se alcanza después de realizar un paso determinado o después de terminada la aplicación del algoritmo.

Pudiera parecer poco importante el análisis de las poscondiciones, pero más adelante se verán ejemplos en los que será necesario tenerlas en cuenta. Por otro lado, se ha dicho antes que la secuencia de pasos del algoritmo tiene que ser lógica y bien definida. Luego, en cierto modo, las poscondiciones de un paso pudieran considerarse como precondiciones del o de los siguientes.

Las poscondiciones establecen una especie de guía para la solución del problema, en el sentido de que se debe tener el control de los estados por los que va pasando la solución del problema desde el momento en que se está desarrollando el algoritmo que lo resuelve.

Ejemplo de poscondición:

El ejemplo de poscondición se basará en la siguiente situación hipotética: supóngase que se desea determinar la situación docente en una asignatura de los estudiantes becados de un grupo. Para ello, habría que promediar las notas en esa asignatura de los becados, exclusivamente. Para calcular el promedio, se analizará, estudiante por estudiante, si es becado. En caso de serlo, se le cuenta como tal y se suma su nota de la asignatura en cuestión. El conteo es necesario para poder calcular el promedio de notas. Luego, pudiera darse el caso que se quiera aplicar ese algoritmo a un grupo en el que no existan becados. En ese caso, la cantidad de becados daría cero pero, para promediar, hay que dividir la suma de las notas entre la cantidad de becados del grupo y no se debe permitir la división por cero.

Para el análisis se tendrá en cuenta sólo el paso a través del cual se cuentan los becados del grupo. La poscondición de ese paso es que la cantidad de becados en el grupo es un número entero no negativo. El hecho de que sea no negativo implica que pudiera ser cero. Es evidente, por tanto, que hay que tener en cuenta las poscondiciones, para evitar que se cometan errores, como el de la división por cero, por ejemplo.

- **Representaciones más comunes**

La representación de un algoritmo establece las estructuras de control para su ejecución. Estas estructuras indican, en cada momento, la operación que se debe realizar para alcanzar la solución del problema planteado.

Existen varias formas de representar los algoritmos. Las más comunes son: el *seudo código* y los *diagramas de bloque*.

La descripción del algoritmo se realizará especificando los siguientes aspectos:

Algoritmo: <Nombre del algoritmo>
El nombre que se utilice para identificar a dicho algoritmo debe ser adecuado y debe expresar en muy pocas palabras lo que hace el algoritmo.

Descripción: <Texto de la descripción>
El texto de la descripción ha de ser breve y en él se deben explicar, en líneas generales, los objetivos y características del algoritmo.

Precondiciones:
- <Precondición 1>
- <Precondición 2>

. . .
- <Precondición n>

Las precondiciones deben especificarse por separado y deben ser claras y precisas.

Poscondiciones:
- <Poscondición 1>
- <Poscondición 2>

. . .
- <Poscondición n>

Las poscondiciones también deben ser expresadas por separado, en un lenguaje claro y preciso.

Entrada:
<Lista de variables de entrada>
Se especifican todas las variables de entrada, o sea, todas aquellas informaciones que necesita el algoritmo para procesar y alcanzar un resultado en la solución al problema.

Salida:
<Lista de resultados>
Se especifican los resultados que se obtendrán con la ejecución del algoritmo. Nótese que no se habla de variables de salida, sino de resultados, ya que el resultado de un algoritmo no tiene que ser siempre un valor. Por ejemplo, un algoritmo para el cálculo de los ceros de un polinomio de grado dos, con ciertos coeficientes, podrá emitir como resultado que no hay raíces reales para los coeficientes utilizados. Eso, sin embargo, es un resultado aunque no se hayan calculado valores resultantes.

Luego de estas especificaciones se detallan los pasos del algoritmo como tal, haciendo uso de alguna de las representaciones antes mencionadas, o sea, mediante un seudo código o un diagrama de bloque.

- **Seudo código**

El *seudo código*, como su nombre lo indica, es una forma de representación que se asemeja al código que se usa para la representación del algoritmo en un lenguaje de programación. Esta forma de representación se basa en el lenguaje natural, o sea, el lenguaje que utilizan los seres humanos para comunicarse entre sí. Esto significa que el seudo código es una *representación literal* de la solución del problema. Tiene pocas reglas sintácticas, por lo que el algoritmo resulta fácil y rápido de desarrollar, y no tiene en cuenta la sintaxis de los lenguajes de programación, por lo que, mediante él, se puede representar casi cualquier operación.

Como el seudo código se expresa en forma literal, cada programador puede inventar el suyo propio. Por ello, mas adelante se muestra una de sus variantes, para que la representación resulte uniforme.
En el próximo epígrafe se verá en detalle cómo escribir un seudo código en la solución de un problema concreto.

- **Diagrama de bloque**

El *diagrama de bloque* es una *representación gráfica* de los algoritmos. Se basa en el uso de figuras geométricas, cada una de las cuales representa una estructura que controla la ejecución del algoritmo.

Las figuras geométricas son enlazadas entre sí por flechas que establecen el flujo de la ejecución del algoritmo, o sea, el orden en que debe ejecutarse sus operaciones o instrucciones. Por lo general, las instrucciones se representan a través de rectángulos.

En el próximo epígrafe se introducen estas estructuras de control.

➢ **Algoritmos lineales y con alternativas**

Luego del estudio de los algoritmos y sus características, de los pasos para resolver un problema, de las precondiciones y poscondiciones y de las representaciones más frecuentes, se puede iniciar el estudio de los *algoritmos lineales* –también conocidos como *algoritmos de secuencia lineal*– y de los *algoritmos con alternativas*.

- **Algoritmos de secuencia lineal**

Los algoritmos, como se ha dicho, son secuencias lógicas de pasos. Para desarrollar su trabajo, necesitan de estructuras de control, que establecen cómo han de aplicarse (*"ejecutarse"*) estos durante el proceso de solución de los problemas. En la ejecución de la secuencia, los pasos se aplican uno tras otro, en el orden en que aparecen en el algoritmo. Ese orden no se puede cambiar durante la ejecución, aún cuando sepamos que no se altera el resultado final al intercambiar dos de los pasos entre sí. En todo caso, lo que se sabrá es que puede haber otra variante para el mismo algoritmo donde aparezcan esos pasos intercambiados, sin que se altere el resultado.

El hecho de poder intercambiar dos pasos entre sí es un aspecto a tener en cuenta a la hora de desarrollar el algoritmo como tal, y el análisis de la posibilidad de intercambio entre ellos dependerá de las precondiciones, de las poscondiciones, de las características del problema y de la variante de solución que se está desarrollando.

Existen algunas estructuras de control que permiten alterar, de manera explícita, el orden de ejecución de los pasos, como se verá más adelante. Cuando no se utiliza este tipo de estructuras, entonces el algoritmo se conoce con el nombre de *algoritmo lineal* o *algoritmo de secuencia lineal*.

Como regla general, los algoritmos de secuencia lineal están formados sólo por instrucciones de asignación y por instrucciones de entrada/salida, a las que se dedicarán los próximos epígrafes.

- **La instrucción de asignación**

Como se explicó antes, todo algoritmo debe presentar una solución general a un problema particular. Para ello, tal como en las matemáticas, la solución a un problema hará uso de variables que tomarán valores durante el proceso de solución.

La *instrucción de asignación*, como su nombre lo indica, se encarga de asignar un valor a una variable. Si, antes de la asignación, la variable tenía un valor, éste es sustituido por el nuevo valor después que se ejecuta esta instrucción.

Se estudiará la sintaxis de las diferentes instrucciones que se utilizan para la solución de los problemas. La *sintaxis* es la forma en que se escribe la instrucción, de manera que tenga un sentido o significado único y entendible por todos. La *semántica* es el significado de la instrucción o la consecuencia de ejecutarla.

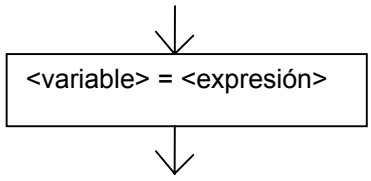
Sintaxis:

- En seudo código:

<variable> = <expresión>

- En el diagrama de bloque:

En el diagrama de bloque la figura que la representa es el rectángulo, según se muestra en el siguiente gráfico:



La instrucción de asignación comienza con la variable a la que se va a asignar el valor, seguida del símbolo de igualdad y, a continuación, se escribe una expresión, que es la que le dará valor a la variable.

Para la descripción de la sintaxis de las instrucciones, se utilizarán diferentes símbolos, como los paréntesis angulares que aparecen a cada lado de las palabras variable y expresión. Estos símbolos se utilizan sólo en la descripción de la sintaxis, para expresar que en ese lugar debe aparecer lo especificado, o sea, en este caso debe aparecer una variable, seguida de un símbolo de igualdad –que se escribe tal cual, puesto que no aparece entre paréntesis angulares– seguido de una expresión. Al escribir la instrucción de asignación se

debe sustituir la frase <variable> de la instrucción por la variable que desea utilizar en ella, sin los paréntesis angulares. Lo mismo sucede con la expresión.

Ejemplos de instrucción de asignación en seudo código:

1. ÁreaCircunferencia = π RadioCircunferencia^2
2. Z = y^3 – h PerímetroRectángulo
3. CantidadDeEstudiantes = 50

En el primero de los ejemplos se asigna a la variable *ÁreaCircunferencia* el valor del producto de π por el cuadrado del valor de la variable *RadioCircunferencia*.

En el segundo ejemplo se resta, al cubo de la variable *y*, el producto de las variables *h* y *PerímetroRectángulo* y el resultado se asigna a la variable *Z*.

El tercer ejemplo muestra la asignación de la constante 50 a la variable *CantidadEstudiantes*.

Los siguientes ejemplos muestran expresiones que no se corresponden con instrucciones de asignación válidas:

1. alfa + beta = 18
2. 15 = Área

Los ejemplos anteriores no son asignaciones válidas porque no aparece una variable en el lado izquierdo del símbolo de igualdad. En el primero de ellos aparece una expresión y, en el segundo, aparece una constante, en el lugar de la variable.

Semántica:

La semántica de la instrucción de asignación es la siguiente:

Primero que todo, se evalúa la expresión de la derecha del símbolo de igualdad teniendo en cuenta las reglas conocidas para ello. Las reglas antes mencionadas pueden ser las de agrupamiento, las de precedencia de los operadores, etc. El valor resultante de dicha evaluación es asignado a la variable que aparece a la izquierda de este símbolo. Esa variable mantendrá el valor asignado hasta que se vuelva a cambiar su valor, ya sea por medio de una asignación o por cualquier otra vía.

Nótese que el sentido de la ejecución de esta instrucción es de derecha a izquierda, o sea, el resultado de la evaluación de la expresión de la derecha del símbolo se asigna a la variable de la izquierda de éste.

Al leer la instrucción de asignación se usa cualquiera de las siguientes frases: variable "recibe" expresión, o bien: a la variable "se le asigna" la expresión. Nunca se utilizará la frase: variable "igual" a expresión. Esto será de esta manera por el hecho de que esta

instrucción no constituye una igualdad matemática, aún cuando en ella se utilice ese símbolo.

Por ejemplo, si se deseara incrementar en uno el valor de la variable *Alfa*, la expresión cuyo resultado daría el nuevo valor es: *Alfa + 1*, y el resultado se quiere en la propia variable *Alfa*, ya que se desea incrementar su valor. Luego, la instrucción de asignación requerida es la siguiente: *Alfa = Alfa + 1*, lo que, de manera obvia, no es una igualdad matemática.

Una última observación en este sentido está relacionada con la importancia de nombrar o identificar de manera adecuada las variables, de forma tal que sea claro el algoritmo o el código que se escribe. De esta forma será más conveniente nombrar a una variable que almacena, por ejemplo, el valor del radio de una circunferencia como *RadioCircunferencia*, y no como *R*, según como se estila en las matemáticas.

- **Instrucciones de Entrada/Salida**

Ya se ha visto cómo asignar valores a las variables. Sin embargo, no siempre los valores de las variables serán asignados, sino que, en ocasiones, los usuarios deben suministrar esos valores. Para ello, se deben utilizar las instrucciones de entrada. Además, se debe contar con instrucciones que permitan la visualización de los resultados del cálculo que ha realizado el algoritmo. Esto se logra a través de las instrucciones de salida. Las instrucciones de entrada/salida posibilitan la comunicación hombre-máquina.

La sintaxis de estas instrucciones se muestra a continuación:

Sintaxis de la instrucción de entrada:

- En seudo código:

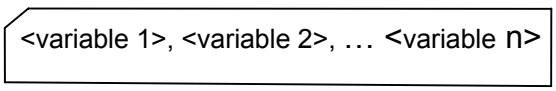
Entrar <variable 1>, <variable 2>, …, <variable n>

Nótese que la palabra Entrar aparece subrayada, para diferenciarla de otras expresiones que se pudieran usar en el algoritmo.

En la práctica, se subrayan las llamadas *palabras reservadas*, las cuales, como su nombre lo indica, son frases cuyo uso está reservado sólo para el propósito con que fueron ideadas. De esta forma no pueden existir variables con esos nombres. Dicho en otras palabras: en ningún algoritmo puede existir una variable cuyo nombre sea *Entrar*, por ejemplo.

- En diagrama de bloque:

La figura que representa a la instrucción de entrada es el rectángulo con la esquina superior derecha cortada, según se muestra en el siguiente gráfico:



Semántica:

En la instrucción de entrada se especifican o listan las variables para las que el usuario suministrará los datos. Se supone que los datos se proporcionan en el orden en que aparecen sus variables correspondientes en la lista de la instrucción.

Sintaxis de la instrucción de salida:

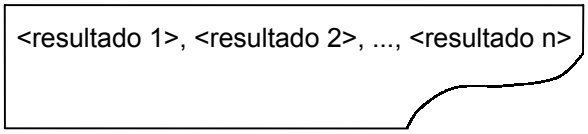
- En seudo código:

Mostrar <resultado 1>, <resultado 2>, …, <resultado n>

Nótese que la palabra Mostrar también es una palabra reservada.

- En diagrama de bloque:

La figura que representa a la instrucción de salida es la que se muestra en el siguiente gráfico:



Semántica:

Esta instrucción solicita la visualización de los resultados listados en el periférico dispuesto para tales efectos.

- **Ejemplo de algoritmo de secuencia lineal**

Para ejemplificar los algoritmos de secuencia lineal se analiza un problema y una variante de solución:

Supóngase que se conoce la capacidad en litros de los dos tanques de almacenamiento de combustible con que cuenta un servicentro. Se sabe que la capacidad de uno de los tanques es menor que la del otro en un por ciento determinado. Por otro lado, se conoce la cantidad total de combustible recibido de los distribuidores en el servicentro. Se necesita saber con cuánto combustible debe llenarse cada tanque, de forma tal que la cantidad de combustible del menor tanque sea menor que la del mayor tanque en el mismo por ciento en que se diferencian sus capacidades.

Nota: Suponga que, inicialmente, los dos tanques están vacíos y que la cantidad de combustible recibida de los distribuidores nunca excederá la suma de las capacidades de los dos tanques.

Para resolver el problema, se seguirán los pasos mencionados anteriormente:

- Comprensión y análisis del problema planteado

Se tienen dos tanques en el servicentro. Uno de ellos es mayor que el otro en un por ciento determinado, o sea, sus capacidades son diferentes en ese por ciento. Se ha recibido una cantidad de combustible y se desea saber con cuánto combustible se debe llenar cada tanque para que queden llenos en la misma proporción de sus capacidades.

- Desarrollo de un modelo matemático que dé solución al problema

Para los valores de las capacidades de ambos tanques, del por ciento en el que difieren esas capacidades, del total de combustible recibido, así como de las cantidades de combustible con que debe llenarse cada tanque, se propone el uso de variables, ya que no se sabe de antemano cuáles serán esos valores. Algunos de ellos serán suministrados por el usuario y otros serán calculados.

Se propone el uso de las variables *CapacidadTanque1* y *CapacidadTanque2*, para almacenar los valores de las capacidades respectivas de los dos tanques; la variable *PorCiento*, para almacenar el por ciento en que se diferencian dichas capacidades; la variable *CombustibleTotal*, en la que se almacenará la cantidad total de combustible recibida; por último, las variables *CombustibleTanque1* y *CombustibleTanque2*, para almacenar las cantidades respectivas de combustible con las que hay que llenar los dos tanques.

Se sabe que la capacidad de uno de los tanques es menor que la del otro en un por ciento determinado, por lo que sólo será necesario conocer la capacidad del mayor de los tanques (valor de la variable *CapacidadTanque1*), ya que la otra capacidad (valor de la variable *CapacidadTanque2*) se puede determinar a partir de ese otro dato. Se necesita conocer, además, el por ciento (valor de la variable PorCiento) en que la capacidad del menor de los tanques difiere de la del otro.

Hasta aquí se tienen dos valores de entrada, o sea, el valor de la variable *CapacidadTanque1* y el de la variable *PorCiento*. Con esos dos datos de entrada se puede calcular el valor de la variable *CapacidadTanque2*.

La capacidad del tanque menor se calcula aplicando una simple regla de tres:

$$CapacidadTanque2 = \frac{PorCiento \cdot CapacidadTanque1}{100}$$

Precondiciones:
- *CapacidadTanque1* debe ser un valor mayor que 0. Lo contrario no es posible.
- *PorCiento* debe ser un valor mayor que 0. De lo contrario el tanque menor tendría una capacidad negativa, lo cual tampoco es posible.

Poscondiciones:
- *CapacidadTanque2* es un valor mayor que 0 y se corresponde con la capacidad del tanque de menor capacidad –o de mayor capacidad, en dependencia de que el por ciento dado sea menor o mayor que 100–, o sea, del segundo tanque.

En este caso, no importa si el por ciento es menor o mayor que 100. Si es mayor que 100, indica que la capacidad del segundo tanque es mayor que la del primero. En caso contrario sería menor.

Una vez que se tiene la capacidad del segundo tanque, se debe determinar la cantidad de combustible con la que se debe llenar cada tanque, de forma tal que para estas cantidades se mantengan las mismas proporciones que para las capacidades. Luego, la cantidad de combustible para el segundo tanque se determina también a través de una simple regla de tres:

$$CombustibleTanque2 = \frac{PorCiento \cdot CombustibleTotal}{100}$$

Precondiciones:
- *CombustibleTotal* debe ser un valor mayor que 0. Lo contrario no es posible.
- *CombustibleTotal* debe ser un valor menor o igual que la suma de las capacidades de los dos tanques.

Poscondiciones:
- *CombustibleTanque2* es un valor mayor o igual que 0 y se corresponde con la cantidad de combustible con la que debe ser llenado el segundo tanque.

Una vez calculada la cantidad de combustible con la que se debe llenar el segundo tanque, se puede determinar la cantidad con la que se debe llenar el primero de ellos, restando del total de combustible la cantidad de combustible del segundo tanque, según:

$$CombustibleTanque1 = CombustibleTotal - CombustibleTanque2$$

Precondiciones:
- No tiene. Esta operación siempre se puede realizar.

Poscondiciones:
- *CombustibleTanque1* es un valor mayor o igual que 0 y se corresponde con la cantidad de combustible con la que debe ser llenado el primer tanque.

Realmente, esta última operación no tiene precondiciones sólo si se han respetado las precondiciones de las operaciones anteriores. De no ser así, no se pudiera garantizar que de la resta resulte un valor mayor o igual que 0.

Este sería el modelo matemático para enfocar la solución al problema. Sin embargo, se puede notar que el valor de la variable *CapacidadTanque2* no se usa, sino sólo para

determinar la capacidad del segundo tanque y ni siquiera es una variable de salida, por lo que esta variable se puede obviar a la hora de desarrollar el algoritmo.

- Desarrollo de la solución del problema mediante un algoritmo

Una vez que se ha desarrollado el modelo matemático, se pasa a elaborar el algoritmo que dará solución al problema planteado, detallando los pasos para la solución del problema. Utilizando los elementos de la descripción planteada anteriormente, el algoritmo sería el siguiente:

Algoritmo: Llenado de tanques en el servicentro

Descripción: Este algoritmo determina la cantidad de combustible con la que debe ser llenado cada uno de los dos tanques en un servicentro, de forma tal que queden llenos con la misma proporción de sus capacidades

Precondiciones:
- La capacidad del primer tanque debe ser mayor que 0.
- El por ciento en que difiere la capacidad del segundo tanque de la del primero debe ser mayor que 0.
- El total de combustible debe ser mayor que 0 y menor o igual que la suma de las capacidades de los dos tanques.

Poscondiciones:
- La cantidad de combustible con la que deben ser llenados los dos tanques son valores mayores o iguales que 0.

Entrada:
- Capacidad en litros del primer tanque (*CapacidadTanque1*)
- Por ciento en que se diferencia la capacidad del segundo tanque de la del primero (*PorCiento*)
- Cantidad total de litros recibidos en el servicentro (*CombustibleTotal*)

Salida:
- Cantidad de combustible con la que debe ser llenado el primer tanque (*CombustibleTanque1*)
- Cantidad de combustible con la que debe ser llenado el segundo tanque (*CombustibleTanque2*)

Seudo código:

Llenado de tanques en el servicentro

Entrar *CapacidadTanque1*, *PorCiento*, *CombustibleTotal*

$$CombustibleTanque2 = \frac{PorCiento \cdot CombustibleTotal}{100}$$

$$CombustibleTanque1 = CombustibleTotal - CombustibleTanque2$$

Mostrar *CombustibleTanque1*, *CombustibleTanque2*
Fin

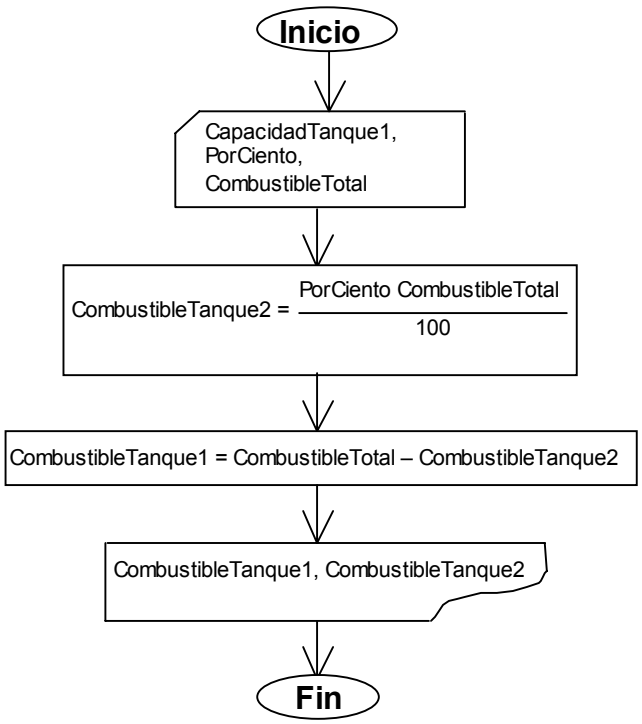
Nótese que se ha comenzado el algoritmo por su nombre y se ha finalizado el mismo con la palabra reservada Fin. Además, es una buena práctica en el desarrollo de algoritmos en seudo código mantener la llamada ***indentación***, la cual consiste en escribir más a la derecha con respecto a la línea anterior del algoritmo cuando la línea actual forma parte de la estructura que se comenzó en líneas anteriores. De esta forma, las cuatro líneas que forman el algoritmo anterior están escritas más a la derecha del nombre y de la palabra Fin, de modo tal que se vea, rápidamente, dónde comienza y dónde termina el algoritmo. Vale aclarar que la indentación sólo tiene relevancia para las formas literales de representación de algoritmos.

Diagrama de bloque:

Se ha dicho anteriormente que, al desarrollar el algoritmo en cualquiera de sus representaciones, se debe especificar el nombre del algoritmo, su descripción, las precondiciones, las poscondiciones, la entrada, la salida y, luego, el algoritmo como tal.

En este caso, sólo se desarrollará el diagrama de bloque y no se especificarán esos otros elementos para no escribirlos dos veces, ya que se especificaron para la representación en forma de seudo código.

Llenado de tanques en el servicentro



Nótese que el diagrama de bloque comienza con un óvalo o elipse con la palabra **Inicio** y finaliza con otro óvalo con la palabra **Fin**.

- **Algoritmos con alternativas**

Se ha dicho antes que la ejecución de los pasos del algoritmo debe efectuarse en el mismo orden en que ellos aparecen en el algoritmo. Sin embargo, existen estructuras de control que posibilitan alterar el orden de la ejecución del algoritmo.

Las *estructuras de alternativa* permiten verificar una o más condiciones y, en dependencia de su valor veritativo, ejecutar unas u otras instrucciones.

Existen dos tipos de alternativas: las simples y las múltiples. Las alternativas simples comprueban sólo el valor veritativo de sólo una condición. Las alternativas múltiples, por su parte, verifican el valor veritativo de varias condiciones.

Para una mejor comprensión, las estudiaremos por separado.

- **Alternativa simple**

La alternativa simple verifica el valor veritativo de una condición determinada y, en dependencia de dicho valor, ejecutará un grupo de instrucciones u otro. Se representa de las siguientes maneras:

Sintaxis:

- En seudo código:

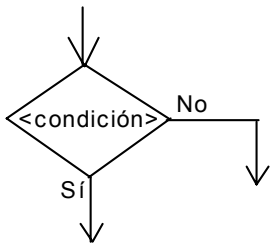
```
Si <condición>
   <conjunto 1 de instrucciones>
[si no
   <conjunto 2 de instrucciones>]
Fin
```

Las expresiones Si, si no y Fin son palabras reservadas. Otro elemento a tener en cuenta es el hecho de que la estructura de alternativa simple termina con Fin. Esto garantiza que en el conjunto 2 se pueda poner cualquier cantidad de instrucciones.

Nótese que la rama del si no aparece entre corchetes, "[" y "]". Esto significa que esta parte de la alternativa es opcional, o sea, se puede poner o no, en dependencia de si es necesaria o no en la solución del problema. Sin embargo, se debe aclarar que el Fin no es opcional. Nótese, además, que de nuevo se hace uso de la indentación.

- En diagrama de bloque:

El rombo es la figura que representa a la alternativa simple:



Semántica:

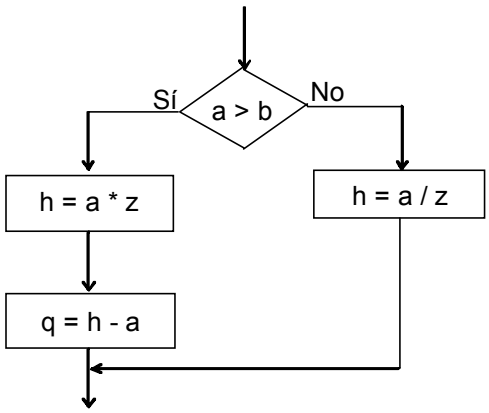
En este caso, se verifica si la condición se satisface o no. En caso afirmativo, se pasa a ejecutar el conjunto de instrucciones que siguen a la condición –en el diagrama de bloque sería la instrucción que aparece por la flecha etiquetada con la palabra "Sí"– y, una vez ejecutadas, se continúa con la instrucción que sigue al Fin. Si no se cumpliera la condición, no se ejecuta el conjunto 1 sino el conjunto 2 de instrucciones –en el diagrama de bloque sería la instrucción que aparece por la flecha etiquetada con la palabra "No"– para, luego, pasar a la instrucción que sigue a la alternativa.

Ejemplo de alternativa simple:

- En seudo código

```
Si a > b
    h = a * z
    q = h - a
si no
    h = a / z
Fin
```

- En diagrama de bloque:



En el ejemplo anterior se averigua si el valor de la variable *a* es mayor que el de la variable *b*. En caso afirmativo, se indica asignar a la variable *h* el producto de los valores de las variables *a* y *z* y a la variable *q*, la diferencia de *h* menos *a*. En caso de que el valor de *a* sea menor o igual que el de la variable *b*, se indica asignar a la variable *h* el valor del cociente de *a* entre *z*. En cualquiera de los dos casos, luego de ejecutar las operaciones correspondientes, se pasa a ejecutar la instrucción que le sigue a la alternativa.

Nótese que la alternativa altera el orden de la ejecución, ya que no se ejecutan las tres instrucciones de asignación, sino que, en dependencia del valor veritativo de la condición, se ejecutan sólo las dos primeras o sólo la tercera.

- **Alternativa múltiple**

La alternativa múltiple verifica el valor veritativo de una condición determinada. Si se satisface, se ejecutan las instrucciones relacionadas con esa condición y, si no se satisface se pasa a probar la condición siguiente siguiendo el mismo criterio. Se representa de las siguientes maneras:

Sintaxis:

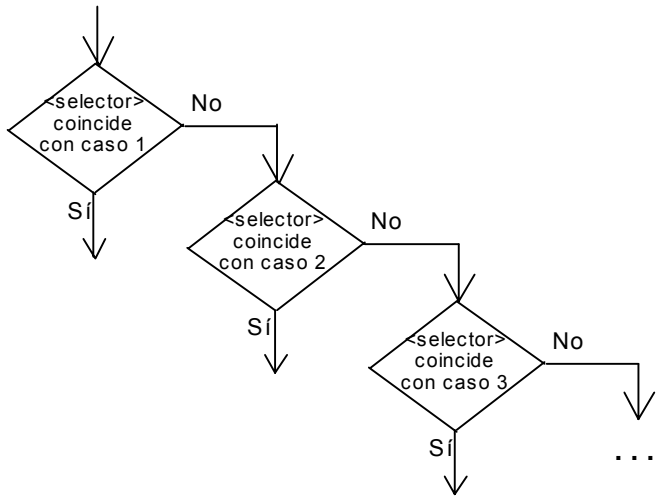
- En seudo código:

En caso que <selector> sea:
<caso 1 >: <conjunto de instrucciones 1>
<caso 2 >: <conjunto de instrucciones 2>
...
<caso n >: <conjunto de instrucciones n>
[si no
<conjunto de instrucciones n + 1>]
Fin

Aquí, las expresiones En caso que, sea y si no también son palabras reservadas. Con el objetivo de que en el conjunto n + 1 de instrucciones se pueda escribir cualquier cantidad de instrucciones, la estructura de alternativa múltiple también termina con Fin.
Nótese que la rama del si no también es opcional, o sea, se puede poner o no, en dependencia de si esta rama es necesaria o no, así como el uso de la indentación.

- En diagrama de bloque:

En el diagrama de bloque la alternativa múltiple se logra anidando varias simples. O sea, que el diagrama de bloque utiliza varios rombos para implementar este tipo de alternativa, según se muestra en el siguiente gráfico:



Semántica:

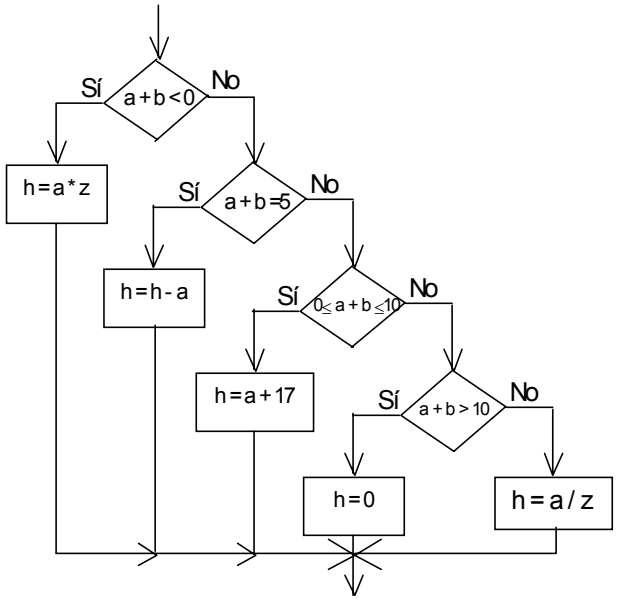
En este caso, el selector es una expresión cualquiera, incluso una variable. Primero, se evalúa este selector para determinar su valor. Luego, se verifica con cuál de los casos especificados casa con ese valor. Si coincide con el caso 1, se ejecuta el primer conjunto de instrucciones, o sea, el conjunto asociado a ese primer caso. Si coincidiera con el segundo, se ejecutará el segundo conjunto de instrucciones y así, sucesivamente. Si en el seudo código el valor del selector no coincidiera con ninguno de los casos especificados y aparece la rama si no de la alternativa, entonces se ejecuta el conjunto n + 1 de instrucciones. En todos los casos, luego de ejecutar el conjunto de instrucciones seleccionado según el caso, se pasa a la instrucción que sigue a la alternativa múltiple.

Ejemplo de alternativa múltiple:

- En seudo código

En caso que a + b sea
    < 0: h = a * z
      5: h = h – a
    ≥ 0 y ≤ 10: h = a + 17
    > 10: h = 0
    si no
        h = a / z
Fin

- En diagrama de bloque:



En el ejemplo anterior se averigua si el valor de *a* + *b* es menor que 0. En caso afirmativo, se indica asignar a la variable *h* el producto de los valores de las variables *a* y *z*. Si no es así, se verifica si dicho valor es igual a 5, en dicho caso se asigna a la variable *h* su valor menos el valor de la variable *a*. Si, por el contrario, el valor de *a* + *b* estuviera entre 0 y 10, se asigna el valor de *a* incrementado en 17 a la variable *h*; si fuera mayor que 10, se le asigna 0 a *h* y, si *a* + *b* no coincide con ninguno de esos casos, se le asigna a la variable *h* el valor del cociente *a* / *z*. En cualquiera de los casos, luego de ejecutar las operaciones correspondientes, se pasa a ejecutar la instrucción que le sigue a la alternativa.

Nótese que la condición de que el valor de *a* + *b* esté entre 0 y 10 se escribe después de la condición de que sea 5. Esto se debe al hecho de que las condiciones se verifican en el mismo orden en que aparecen en la alternativa múltiple. Debe respetarse ese orden, ya que, si esas dos condiciones se intercambian, a *h* nunca se le asignará *h* – *a*, ya que el valor 5 también cumple la condición de estar entre 0 y 10. Por tanto, hay que tener cierto cuidado con el orden a la hora de escribir las condiciones de una alternativa múltiple.

➢ Modularidad. Procedimientos y funciones. Representación

Ya hemos visto cómo representar un algoritmo en las dos formas más frecuentes: la literal –en forma de seudo código– y la gráfica –en forma de diagrama de bloque. Ahora se estudiará otro aspecto crucial en la algoritmización de la solución de los problemas: *la modularidad*.

• Modularidad

La *modularidad*, definida de manera informal, es una técnica de algoritmización que permite al programador diseñar o desarrollar los módulos que intervienen en la solución de los problemas.

Los *módulos* son segmentos de algoritmos independientes y relativamente autónomos que desarrollan una tarea específica y son invocados en el momento en que se necesite de su ejecución.

Se dice que son **independientes**, ya que, por lo general, el programador los desarrolla cada uno por separado. Son **relativamente autónomos** porque realizan solos su labor, o sea, sin la ayuda de otros módulos. Cada módulo **resuelve una tarea específica**: aquella para la cual fue diseñado. Después que son diseñados, los módulos quedan disponibles para ser **invocados** –utilizados– **en el momento necesario** de la solución del problema.

Un módulo será más útil en la medida en que sea más general, o sea, en la medida en que pueda ser aplicado a una gama mayor de problemas. Por ejemplo, se pudiera diseñar un módulo para multiplicar dos valores numéricos enteros y otro para multiplicar dos valores numéricos reales. Sin embargo, en la mayoría de los casos es más general desarrollar un módulo que multiplique dos valores numéricos, independientemente de su naturaleza. De esta forma, el módulo servirá, tanto para valores enteros como para reales.

Vale aclarar que desde un módulo se puede invocar a otros, e incluso a él mismo.

Ventajas del uso de la modularidad:

El uso de la modularidad proporciona grandes ventajas, algunas de las cuales se exponen a continuación:

- Durante el desarrollo del módulo, el diseñador se concentra sólo en la tarea que debe realizar dicho módulo, dejando el análisis de otras tareas para otro momento.
- Se logra *portabilidad*, o sea, que el módulo pueda ser invocado para resolver tareas similares en otros algoritmos. La portabilidad implica, además, que un diseñador haga uso de un módulo desarrollado por otro para la solución de otros problemas.
- En la solución de un mismo problema el módulo se diseña una sola vez y puede ser invocado todas las veces que sea necesario.
- A la hora de invocar un módulo, el diseñador no necesita conocer cómo éste está implementado. En realidad, en ciertas ocasiones esto constituye una desventaja.
- La modularidad permite que el problema sea dividido en diferentes partes, cada una de las cuales se resuelve a través de uno o varios módulos. La combinación adecuada de los resultados de la invocación de cada uno de ellos proporcionará la solución al problema original.

A través de la modularidad se ha impuesto un estilo de programación mediante el cual el diseñador plantea la solución al problema en términos de tareas generales y, luego, cada tarea se programa en módulos que puede ser, a su vez, general, por lo que se puede seguir subdividiendo y así, sucesivamente, hasta que la tarea que se analiza no se pueda subdividir.

Por el momento se estudiarán dos tipos de módulos: las funciones y los procedimientos.

- **Funciones**

Las funciones son tipos de módulos que devuelven como resultado un valor. De las matemáticas se conocen ejemplos de funciones: las trigonométricas (seno, coseno, tangente, etc.), las logarítmicas (logaritmo natural, logaritmo de base 10, etc.), la exponencial (e^x), etc.

En los lenguajes de programación se cuenta con una amplia colección de funciones, conocidas con el nombre de *funciones estándar* o *funciones suministradas*. Ahora bien, las funciones antes mencionadas son funciones ya implementadas. De lo que se trata aquí es de implementar funciones que no existen, para dar solución a los diferentes problemas.

Las funciones son invocadas a través de su nombre y siempre como parte de una expresión.

- **Procedimientos**

Los procedimientos son otro tipo de módulo que no devuelven ningún valor, sino que son diseñados para realizar una tarea. Un ejemplo de procedimiento lo constituye un módulo

que borre la pantalla de la computadora. Como es evidente, este módulo no devuelve ningún valor, ya que borrar la pantalla no tiene ningún valor resultante.

En los lenguajes de programación también se cuenta con una amplia colección de *procedimientos estándar* o *procedimientos suministrados*. Aquí, también, se requiere del diseño de procedimientos que no existen para resolver nuevos problemas.

Los procedimientos se invocan a través de su nombre y esta invocación constituye una instrucción.

- **Representación**

La representación de las funciones y procedimientos se enfoca en las dos líneas siguientes:

- la de la definición, desarrollo o diseño de la función o el procedimiento, o sea, cuál es la sintaxis con la que se define la función o el procedimiento, y
- la de la invocación, aplicación, llamada o utilización de la función o el procedimiento, o sea, cuál es la sintaxis de su invocación en un algoritmo.

Para una mejor comprensión, se exponen la sintaxis de la definición y luego de la invocación; primero de la función y, por último, del procedimiento. La definición de módulos tipo función y procedimiento en diagrama de bloque no tiene diferencia ninguna con la de un algoritmo cualquiera, por lo que se estudiará sólo la sintaxis de la definición en seudo código.

- *Definición de la función*

Sintaxis:

Función <nombre de la función> [(<lista de parámetros formales>)]
  <cuerpo de la función>
Fin

La función se identifica por su nombre y, a continuación, se escribe, entre paréntesis, la <lista de parámetros formales>. Esta lista es opcional, o sea, se pondrá sólo si es necesaria y servirá para la transferencia de información hacia y desde el módulo en cuestión. El <cuerpo de la función> es la secuencia de instrucciones que desarrollan la tarea para la que fue diseñada la función y debe tener, al menos, una instrucción donde al nombre de la función se le asigna el valor resultante. Esto es, porque se ha dicho que la función devuelve un valor y lo devuelve a través de su nombre.

La lista de parámetros formales se denomina de esa forma, ya que con sus elementos se formaliza la transferencia de información mencionada anteriormente. La función necesitará en ocasiones datos o informaciones y se les suministrarán a través de la lista de parámetros formales. La función, en ocasiones, retornará otros datos, además del que devuelve por ser función y los proporcionará a través de la lista de parámetros formales. En ocasiones, incluso, la función recibirá un dato, lo procesará, alterando su valor, y devolverá el valor alterado.

Este razonamiento lleva a clasificar los parámetros de la lista en tres grupos:

- *Parámetros de entrada*: Son datos que se suministran al módulo (ya sea función o procedimiento). El módulo utiliza su valor para poder dar solución a la tarea para la que se diseñó dicho módulo. Sin ese dato no podrá trabajar.
- *Parámetros de salida*: Son datos que resultan del procesamiento del módulo (ya sea función o procedimiento) y que se devuelven para ser utilizados posteriormente. No se deben confundir con el valor que devuelve una función, ya que la función devuelve su valor a través de su nombre como se ha indicado anteriormente y esta dato se devuelve a través de la lista de parámetros formales. En el caso de las funciones los parámetros de salida se utilizan para que la función pueda devolver más de un dato.
- *Parámetros de entrada – salida*: Son datos que se suministran al módulo y éste los procesa, altera su valor y devuelve su valor alterado.

Ejemplo de definición de función:

En el siguiente ejemplo se define una función para determinar el mayor de dos valores numéricos cualesquiera.

La función es un módulo o segmento de algoritmo, por lo que seguiremos los elementos antes mencionados para su descripción:

- Comprensión y análisis del problema:

Dados dos valores, se debe determinar el mayor de ellos.

- Desarrollo de un modelo matemático que dé solución al problema

Los dos valores deben compararse entre sí. Si el valor del primero es mayor que el del segundo, entonces se debe devolver el valor del primero. Si, por el contrario, el valor del segundo es el mayor de los dos, se debe devolver este valor.

- Desarrollo de la solución del problema mediante un algoritmo

En primer lugar, la función necesita de los dos valores, luego hay que pasarle dos parámetros. Ambos parámetros son sólo de entrada, ya que suministrarán a la función los valores a comparar. La función debe devolver el mayor de ellos, es decir, que el valor que devuelve la función es el mayor de los dos valores.

Algoritmo: Mayor de dos números

Descripción: Este algoritmo determina el mayor entre dos valores numéricos cualesquiera.

Precondiciones:
- Los dos valores son numéricos.

Poscondiciones:
- El resultado es el mayor entre los dos valores de entrada

Entrada:
- Valores a y b

Salida:
- El mayor de los valores

```
          ⇩⇩
Función Mayor(a, b)
  Si a > b
    Mayor = a
  si no
    Mayor = b
  Fin
Fin
```

La función anterior se llama Mayor. Es necesario identificar las funciones con nombres adecuados, como se dijo anteriormente para las variables. Los dos parámetros formales son de entrada, lo cual se ha especificado con una flecha cuya saeta señala al parámetro en cada caso. El cuerpo de la función está formado por una alternativa que decide cuál de los valores de las variables, a o b, es el mayor. Nótese que al nombre de la función, *Mayor*, se le asigna el mayor de los valores. Al invocar la función se devolverá el mayor de los valores en el nombre de la función.

- *Invocación de la función*

Como se dijo anteriormente, la función se invoca como parte de una expresión. Es por ello que se devuelve el resultado a través de su nombre.

Sintaxis:

- En seudo código:

... <nombre de la función> [(<lista de parámetros actuales>)] ...

Los puntos suspensivos indican que la llamada o invocación a la función no aparece sola, sino que se escribe como parte de una expresión. La lista de parámetros es opcional, pero si la función se definió con parámetros, la invocación debe llevar parámetros actuales. Por supuesto que *cada parámetro actual de la invocación se corresponde con un parámetro formal en la definición*.

Aquí los parámetros son actuales, o sea, son los parámetros con los que actualmente se invoca a la función.

Ejemplo de invocación de función:

Haciendo uso de la función definida en el ejemplo anterior, la invocación pudiera ser como sigue:

Z = Mayor(h, 50)

En el ejemplo se hace uso de la función *Mayor* para determinar el mayor entre el valor de la variable *h* y la constante 50. La función devuelve ese valor y la instrucción de asignación lo asigna a la variable *Z*. Nótese que la invocación se realiza como parte de una expresión: la correspondiente a la asignación.

- *Definición del procedimiento*

Sintaxis:

Procedimiento <nombre del procedimiento> [(<lista de parámetros formales>)]
  <cuerpo del procedimiento>
Fin

El procedimiento se identifica por su nombre y, a continuación puede aparecer opcionalmente, entre paréntesis, la <lista de parámetros formales>, cuyo objetivo es la transferencia de información hacia y desde este módulo. El <cuerpo del procedimiento> es la secuencia de instrucciones que desarrollan la tarea para la que fue diseñado el procedimiento. En este caso, como el procedimiento no devuelve ningún valor, no puede haber ninguna instrucción donde al nombre del procedimiento se le asigna valor resultante alguno.

La lista de parámetros formales tiene las mismas características que para las funciones.

Ejemplo de definición de procedimiento:

En el siguiente ejemplo se definirá un procedimiento que incrementa un valor dado en un valor de incremento determinado.

- Comprensión y análisis del problema:

Dado un valor, se debe incrementar el mismo en un valor de incremento determinado.

- Desarrollo de un modelo matemático que dé solución al problema

El valor debe ser incrementado y este incremento debe ser almacenado en la propia variable.

- Desarrollo de la solución del problema mediante un algoritmo

Este procedimiento necesita de dos valores: el valor a incrementar y el valor del incremento. Ambos valores son de entrada. Sin embargo, el valor a incrementar es también un parámetro de salida, ya que es importante su valor después de ser alterado.

Algoritmo: Incremento de un valor en un valor de incremento dado

Descripción: Este algoritmo incrementa el valor de una variable en un incremento determinado.

Precondiciones:
- Los dos valores son numéricos.

Poscondiciones:
- El valor de la variable está incrementado en el incremento dado

Entrada:
- Variable
- Valor de incremento

Salida:
- Número incrementado

```
Procedimiento Incrementa(Número, incremento)
  Número = Número + incremento
Fin
```

El procedimiento se llama *Incrementa*. El primer parámetro formal es de entrada-salida y el segundo es sólo de entrada, lo cual se ha especificado con las flechas cuya saeta indican entrada y/o salida. El cuerpo del procedimiento está formado por una instrucción de asignación que incrementa el valor de la variable *Número*.

- *Invocación del procedimiento*

Como se dijo anteriormente, el procedimiento se invoca por su nombre y esta invocación constituye una instrucción.

Sintaxis:

- En seudo código:

<nombre del procedimiento> [(<lista de parámetros actuales>)]

La lista de parámetros actuales es opcional, pero si el procedimiento se definió con parámetros, la invocación debe llevar parámetros actuales. También para los procedimientos, *cada parámetro actual de la invocación se corresponde con un parámetro formal en la definición*.

Ejemplo de invocación del procedimiento:

Haciendo uso del procedimiento definido en el ejemplo anterior, la invocación pudiera ser como sigue:

Incrementa(h, 50)

En el ejemplo se hace uso del procedimiento *Incrementa* para incrementar el valor de la variable *h* en la constante 50. Nótese que la invocación constituye una instrucción como tal.

Es necesario aclarar que al diseñar un módulo, lo primero que hay que decidir es si éste es una función o un procedimiento.

- **Ejecución paso a paso**

La ejecución paso a paso es la aplicación del algoritmo a un juego de datos concreto, o sea, a una entrada específica. Esta aplicación ejecuta cada instrucción del algoritmo, permitiendo determinar, en primer lugar qué hace el algoritmo y cómo lo hace.

Sabiendo lo que hace un algoritmo se pueden detectar, a través de la ejecución paso a paso, posibles errores de programación que pueda presentar éste. Por otro lado, el diseñador pudiera darse cuenta de algunas modificaciones que pudieran hacerse para que el algoritmo trabaje más rápido o utilice menos memoria, etc.

Es por ello que la ejecución paso a paso es importante

- **Ejemplos**

1. En el primer ejemplo se estudiará un módulo para la determinación del mayor de tres valores.

   - Comprensión y análisis del problema

   El problema consiste en determinar el mayor de tres valores, es decir, conocidos los valores se debe determinar cuál, en magnitud, es el mayor de ellos.

   Este caso es similar al visto anteriormente, sólo que ahora se tienen tres en lugar de dos valores.

   - Desarrollo de un modelo matemático que dé solución al problema

   Se deben comparar los tres valores entre sí para determinar cuál de ellos es el mayor.

   En aras de ganar en generalidad, se utilizará una variable auxiliar, en la que se almacene el mayor de los dos primeros. Luego, ese valor se compara con el tercero y se determina así el mayor de los tres.

- Desarrollo de la solución mediante un algoritmo

¿Qué tipo de módulo utilizar? ¿Procedimiento o función?. En este sentido, se debe devolver el mayor de los tres valores, ya que no está dicho para qué es necesario ese dato en cálculos posteriores. Por lo tanto, lo más adecuado es una función.

Esta función debe recibir los tres valores a comparar. Eso implica que los tres valores son de entrada. En este caso no hay parámetros de salida, pues el valor a devolver es uno solo.

Algoritmo: Mayor de tres valores dados

Descripción: Este algoritmo determina el mayor de tres valores y lo devuelve para su posterior uso.

Precondiciones:
- Los tres valores son numéricos.

Poscondiciones:
- El mayor de los tres también es un valor numérico

Entrada:
- Tres valores

Salida:
- Mayor de los tres valores

```
                          ⇩       ⇩       ⇩
Función MayorDeTres(Valor1, Valor2, Valor3)
  Auxiliar = Valor1
  Si Valor2 > Auxiliar
        Auxiliar = Valor2
  Fin
  Si Valor3 > Auxiliar
     Auxiliar = Valor3
  Fin
  MayorDeTres = Auxiliar
 Fin
```

La función se denomina *MayorDeTres*. Los tres parámetros se corresponden con los tres valores a comparar y son de entrada. Primero se asigna a la variable *Auxiliar* el valor del primero de los tres valores. Luego se compara el valor del segundo con el de *Auxiliar*. Si el de *Valor2* es mayor, se almacena en la variable *Auxiliar*, con lo que se garantiza que esta variable tenga almacenado el mayor de los dos primeros valores. A continuación, se compara el valor de la variable *Auxiliar* con el del parámetro *Valor3*. Si este valor es mayor que el de *Auxiliar*, entonces se asigna éste a *Auxiliar*, con lo que se garantiza que esta variable tenga almacenado el mayor de los tres valores de

entrada. Por último se asigna, al nombre de la función, el de la variable *Auxiliar*, lo cual prepara a la función para que cuando sea invocada o llamada con tres valores cualesquiera, devuelva el mayor de ellos.

Las estructuras repetitivas son la base de los algoritmos básicos que se estudiarán, aunque se conjugan con las alternativas y las variables o colecciones de valores.

- Suma iterada

La suma iterada es un proceso de suma de varios valores conocidos, de los que se puede conocer o no su cantidad.
Este proceso de suma de valores numéricos se desarrolla siguiendo una idea bastante sencilla que utiliza una variable auxiliar para realizar la suma.
La idea es almacenar en la variable donde se va a depositar la suma el primero de los valores a sumar. El segundo valor se suma al valor de la variable sumadora y el resultado de dicha suma se deposita de nuevo en la variable sumadora y así, sucesivamente. Esto se repite para todos los valores que se desean sumar.
Ahora bien, la idea anterior plantea que el primer valor se trate de modo diferente que los demás, o sea, que se almacene en la variable sumadora el primer valor y luego se itere la suma. Sin embargo, se puede lograr hacer lo mismo iterando toda la suma. Esto, sin embargo, implica que la instrucción a repetir sea la asignación a la variable sumadora de su valor incrementado en el número que se desea sumar en este momento. Para garantizar, entonces, que el resultado de la suma no sea incorrecto, se debe inicializar la variable sumadora con el valor 0, que es el neutro de la suma.

Por tanto, el proceso de suma iterada consta de los siguientes momentos:

1. Inicialización de la variable sumadora con el neutro de la suma (antes del ciclo).
2. Repetición o iteración del incremento de la variable sumadora con todos los valores a sumar (durante el ciclo).
3. Obtención del resultado de la suma iterada (después del ciclo).

- Conteo

El conteo es un caso particular de suma iterada, en la que el incremento de la variable contadora es en un valor fijo y no en un valor variable como en el caso de la suma iterada. Esto significa que a la variable contadora, o simplemente contador, se le suma en el segundo momento una cantidad fija.
Al ser un caso particular de suma iterada, el conteo también pasa por los tres momentos antes mencionados.

- Producto iterado

El producto iterado sigue una idea similar a la utilizada en la suma iterada, sólo que ahora la operación es de multiplicación y la variable del producto se inicializa con el valor 1 que es el neutro del producto, en lugar de 0 como en la suma.

El proceso del producto iterado tiene, también, tres momentos:

1. Inicialización de la variable producto con el neutro de la multiplicación (antes del ciclo).
2. Repetición o iteración del producto con todos los valores a multiplicar (durante el ciclo).
3. Obtención del resultado del producto iterado (después del ciclo).

Un ejemplo de producto iterado lo constituye el cálculo del factorial de un número entero no negativo.

➢ Algoritmos con ciclos

En la práctica, aparecen con frecuencia situaciones en las que se necesita que un conjunto de instrucciones se ejecute repetidamente. En este caso se debe contar con estructuras de control que permitan la iteración o repetición de dicho conjunto de instrucciones. Estas estructuras deben permitir la verificación de condiciones que determinan si el conjunto de instrucciones debe seguir siendo repetido o si debe detenerse dicha repetición o iteración.

- **Tipos de ciclos**

Existen distintos tipos de ciclo, los cuales serán utilizados en dependencia de la situación concreta que se desea resolver.

Todos los tipos de ciclos verifican, de alguna manera, el cumplimiento de una condición para determinar si el ciclo debe detenerse o no.

En ocasiones se conoce la cantidad de veces que el ciclo debe ser repetido y en otras no. Teniendo en cuenta este criterio, los ciclos pueden clasificarse en:

- Ciclos por condición
- Ciclos por variable de control

Para una mejor comprensión, los explicaremos por separado.

▪ Ciclos por condición

En estos ciclos aparece la condición de manera explícita, o sea, se escribe la condición que hay que verificar como parte de la sintaxis de estas estructuras repetitivas.

La condición a verificar es una expresión booleana cualquiera que determina cuándo el ciclo debe detenerse y puede verificarse antes de ejecutarse la secuencia de instrucciones o después de dicha ejecución. Se le llama expresión booleana a una expresión lógica que puede tomar valor verdadero o falso y su cumplimiento o no, determina la verificación o no de la condición del ciclo.

De acuerdo al momento en que se verifica el cumplimiento de la condición, con respecto a la ejecución del conjunto de instrucciones, este tipo de ciclos se puede subdividir, a su vez, en:

- Ciclo con precondición.
- Ciclo con poscondición.

- **Ciclo con precondición**

Como su nombre lo indica, en este ciclo la verificación de la condición ocurre antes de la ejecución de la secuencia de instrucciones que se desea iterar.

En seudo código, esta instrucción tiene la sintaxis siguiente:

Mientras <condición>
  <secuencia de instrucciones>
Fin
Primero que todo, se evalúa la expresión booleana que constituye la condición. Se verifica su cumplimiento y, si se satisface, se ejecuta la secuencia de instrucciones una primera vez. Al finalizar esta ejecución, se vuelve a evaluar y verificar la condición y, si se satisface, se ejecuta de nuevo la secuencia de instrucciones y así, sucesivamente. La primera vez que no se cumple la condición, se sale de la estructura repetitiva y se continúa con la instrucción que le sigue.
Nótese que, al evaluar por primera vez la condición, puede suceder que no se cumpla. En ese caso, la secuencia de instrucciones no se ejecuta nunca.
Es importante aclarar que en este tipo de ciclo, para evitar que la ejecución sea infinita, la condición tiene que dejar de cumplirse en algún momento. Para garantizar lo anterior, la secuencia de instrucciones a iterar debe alterar en algún momento la validez de la condición, o sea, debe contar con, al menos, una instrucción que cambie el sentido de dicha condición.

- **Ciclo con poscondición**

Como su nombre lo indica, en este ciclo la verificación de la condición ocurre después de la ejecución de la secuencia de instrucciones que se desea iterar.

En seudo código, esta instrucción tiene la sintaxis siguiente:

Repetir
  <secuencia de instrucciones>
hasta <condición>
Primero que todo, se ejecuta la secuencia de instrucciones. Luego, se evalúa la expresión booleana que constituye la condición. Se verifica su cumplimiento y, si **no** se satisface, se repite nuevamente la secuencia de instrucciones. Al finalizar esta ejecución, se vuelve a evaluar y verificar la condición y, si **no** se satisface, se ejecuta de nuevo la secuencia de instrucciones y así, sucesivamente. La primera vez que se cumple la condición, se sale de la estructura repetitiva y se continúa con la instrucción que le sigue.

Nótese que, al evaluar por primera vez la condición, puede suceder que se cumpla. En ese caso, la secuencia de instrucciones se ejecuta completa al menos una vez.
Es importante aclarar que en este tipo de ciclo, para evitar que la ejecución sea infinita, la condición tiene que satisfacerse en algún momento. Para garantizar lo anterior, la secuencia

de instrucciones a iterar también debe contar con, al menos, una instrucción que cambie el sentido de la condición.

- **Ciclos por variable de control**

En estos ciclos aparece la condición de manera implícita, o sea, **no** se escribe condición alguna que haya que verificar como parte de la sintaxis de estas estructuras repetitivas. Estos ciclos se caracterizan por utilizar una variable de control que determina la cantidad de veces que se debe repetir la secuencia de instrucciones. Esta es la razón por la que se les conozca, también, como ciclos por conteo o por contador. En este caso, la condición la determina el hecho de que la variable de control haya alcanzado, o no, el valor correspondiente a la cantidad de veces que hay que repetir la secuencia de instrucciones.

Esto significa que este tipo de ciclo se utiliza cuando es conocida la cantidad de veces que se debe repetir la secuencia de instrucciones.

Sintaxis del ciclo por variable de control:

Para <var. de control> = <v. inicial> hasta <v. final> [con paso <incremento>]
  <secuencia de instrucciones>
Fin

donde:
<var. de control> es la variable de control del ciclo
<v. inicial> es el valor inicial de la variable de control
<v. final> es el valor final de dicha variable
<incremento> es el valor con el que se incremente la variable de control del ciclo

Tanto el valor inicial como el valor final y el incremento pueden ser valores constantes, variables o expresiones.

Nótese que la especificación del incremento del valor de la variable de control es opcional.

Primero que todo, se determina el valor inicial, el final y el de incremento de la variable de control del ciclo. Eso significa que si alguno de ellos es una expresión, ésta se evalúa; si alguno es una variable, se toma su valor y si alguno es una constante, entonces se toma su valor directamente.
Una vez que se han determinado estos tres valores se inicializa la variable de control del ciclo con el valor inicial y se compara con el valor final. Si la variable de control no ha sobrepasado el valor final, se ejecuta la secuencia de instrucciones. Luego, se incrementa la variable de control con el valor de incremento especificado con el paso. Si no se especifica paso alguno, se supondrá que la variable de control se incrementa en 1. Luego del incremento, se vuelve a verificar si la variable de control ha sobrepasado el valor final. Si no es así, se repite el proceso, hasta que la variable de control sobrepasa el valor final. La primera vez que esto ocurre, se sale de la estructura repetitiva y se continúa la ejecución con la instrucción siguiente.

Es posible que sea necesario que el valor inicial sea mayor que el final. En ese caso, el incremento debe ser negativo. De no ser así, el ciclo sería infinito.

- Cálculo de valores extremos

Otra de las operaciones básicas frecuentes en la solución de problemas de la Informática es la determinación de los valores extremos de un conjunto de datos.

Tema IV: Introducción a la programación

El tema que comienza, se ocupa del estudio de elementos básicos de programación que permitirán en lo adelante escribir programas para ser ejecutados en una computadora. Se hará énfasis en los algoritmos básicos estudiados anteriormente y su representación en un lenguaje de programación.
Para ello se estudiarán sentencias básicas del lenguaje de programación y la sintaxis de cada una se analizará en el marco del lenguaje Object Pascal que ha sido el lenguaje escogido para la introducción de los conceptos de programación.

1.- Conceptos básicos

Lenguaje de programación

Un lenguaje de programación es aquel que se utiliza para escribir algoritmos que serán ejecutados en computadoras, por lo que puede considerarse un lenguaje intermediario que permite determinada comunicación entre el hombre y la computadora. De una manera informal se puede definir como un conjunto de reglas sintácticas, o sea reglas de producción, que permiten formar sentencias con un valor semántico asociado según la gramática definida para el lenguaje. Todo lenguaje de programación define su propio alfabeto, cuyos símbolos son caracteres alfanuméricos o cadenas de éstos. Las sentencias se construyen a partir de proposiciones simples y/o compuestas.
Los lenguajes de programación han evolucionado desde lenguajes de máquinas hasta superlenguajes o lenguajes de alto nivel. Los primeros son lenguajes de muy bajo nivel, más cercanos a las máquinas, en los que se escriben instrucciones muy simples y que resultan muy incómodos para el hombre, por ejemplo Ensamblador. Los últimos son lenguajes que resultan más comprensibles y fáciles de usar por los hombres pues utilizan algunos términos y reglas sintácticas de los lenguajes naturales para la construcción de instrucciones.

Programa

En el mundo de la computación se conoce como programa a todo conjunto de órdenes que pueden ser ejecutadas por una computadora produciendo determinado comportamiento de ésta. Un programa no es más que un algoritmo transcrito en cierto lenguaje de programación y dirigido a una computadora.
Toda la comunicación de los usuarios con las computadoras se produce gracias a la ejecución de algún programa y a través de éste.
Como consecuencia del aumento lógico de la complejidad de los programas y del cambio de paradigma de interacción hombre-máquina, el arte de la programación, o sea de la escritura de programas para computadoras ha transitado por diferentes etapas caracterizadas por el estilo asumido al abordar el diseño de éstos. La etapa de ***programación monolítica***, se caracterizó por un diseño de programas considerados como un todo, los cuales eran mejores mientras más sofisticados eran, más tareas realizaran y más memoria ahorraran. Optimizar era una señal de eficiencia. Se utilizaban los diagramas de bloques como herramientas para representar los algoritmos. La programación monolítica era incapaz de dar respuesta al crecimiento de los programas tanto en tamaño como en complejidad. A esta le siguió, por tanto, una nueva etapa conocida como de ***programación modular o***

estructurada y que puede resumirse en un estilo de diseño de programas basado en la técnica de divide y vencerás, a partir del cual un programa puede concebirse como un conjunto de módulos cada uno de los cuales implementa una función determinada, tiene una sola entrada y una sola salida y puede ser probado independientemente. Para representar los algoritmos a implementar por los diferentes módulos se utilizaban todavía los diagramas de bloques. Este nuevo estilo de programación, aunque no era la respuesta final, permitió aumentar el personal en los equipos de programación y contribuyó al ahorro de tiempo y recursos. Para el diseño modular la piedra angular son las tareas a desarrollar para solucionar el problema, cada una de las cuales puede desempeñarse en un modulo de relativa independencia. Entre estos módulos viajan los distintos datos necesarios para resolver cada tarea concreta de manera que a cada uno llegan solamente aquellos que son necesarios. Otra forma de abordar la programación es con un enfoque más centrado en la modelación de los objetos del problema a resolver, o lo que es lo mismo los datos y las operaciones sobre éstos, la cual ha dado lugar a una etapa marcada por un diseño de ***programas orientados a los objetos***. Este estilo de abordar la programación no echa por tierra los principios de la programación modular sino que los complementa elevando ésta a un nivel cualitativamente superior. Por último, el diseño orientado a objetos ha dado lugar al surgimiento de las interfaces gráficas de usuarios y esto a su vez ha dado lugar a un nuevo estilo de programación denominado ***programación dirigida a eventos*** que centra la atención del diseño de los programas en la atención a los distintos eventos que pueden ser generados durante la ejecución de una aplicación y que pueden ser originados por los usuarios, otras aplicaciones y los propios objetos.

Se dice que un lenguaje de programación soporta un determinado enfoque o estilo de programación, si cuenta con herramientas que garantizan y velan por la implementación adecuada de las estructuras básicas de dicho estilo. Así por ejemplo, se dice que Fortran (en sus últimas versiones), Basic, Quick Basic, Pascal y C (estos últimos en algunas de sus versiones iniciales) son lenguajes estructurados, Smalltak es un lenguaje orientado a objetos puro, otros como Turbo Pascal 5.5 y superiores, Borland Pascal, Object Pascal y C++ son lenguajes híbridos pues soportan más de un enfoque.

Existen otros estilos de programación aunque no se reconocen etapas caracterizadas por el uso de dichos estilos, pues nunca se han usado de manera masiva por los programadores, ni para abordar cualquier tipo de problemas, sino un grupo reducido de éstos. Este es el caso de la programación descriptiva que incluye la programación lógica (soportada por lenguajes como Prolog) y la programación funcional (soportada por lenguajes como Lisp).

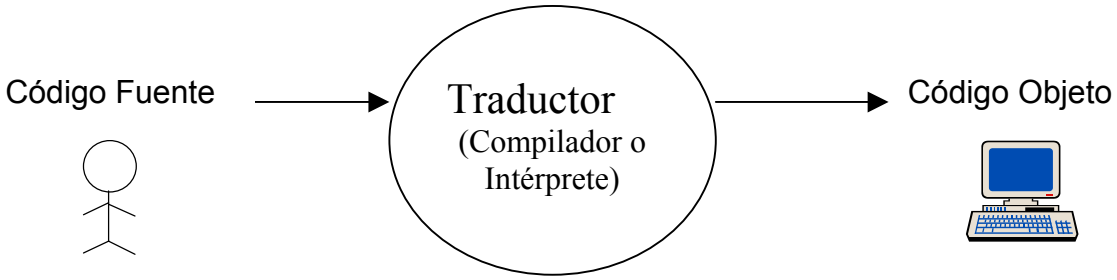
Aplicación

Gracias a la evolución del hardware y el software, se construyen hoy en día programas o sistemas de computadoras que resuelven problemas cada vez más complejos y donde se procesan generalmente grandes volúmenes de información. Dichos sistemas pueden verse por su nivel de complejidad y estructuración como un conjunto de programas más simples relacionados entre sí, por lo que se prefiere utilizar el término aplicación para acuñarlos. No obstante es muy común el uso de los términos programas y aplicación indistintamente.

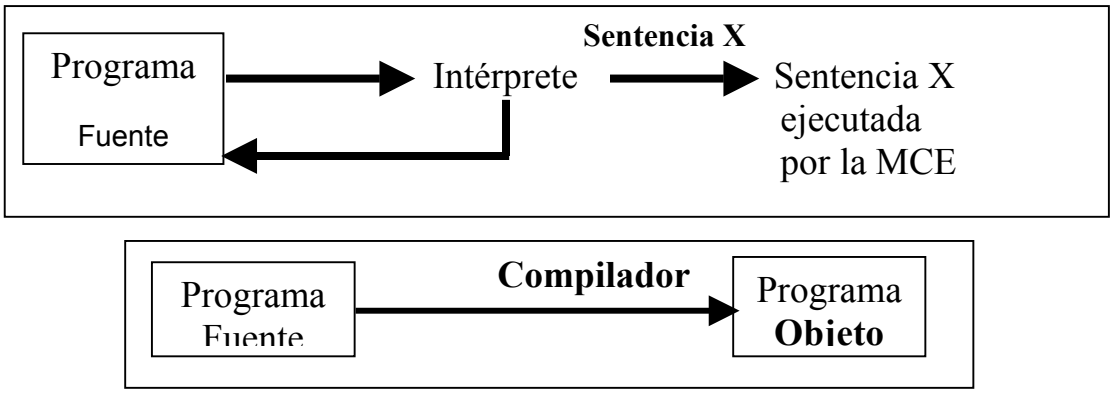
Código fuente
A los programas escritos en lenguajes de programación se les llama código fuente. O sea, el hombre escribe código fuente.

Código objeto
Los programas en código fuente son comprensibles por el hombre pero no son directamente comprensibles por la máquina. Para que esto suceda, o sea, para que el código fuente pueda ser entendido y ejecutado por la computadora, es necesario traducirlo a código objeto, que es el código de máquina.

Compiladores e intérpretes
El proceso de traducción del código fuente al código objeto se puede realizar de diferentes maneras.

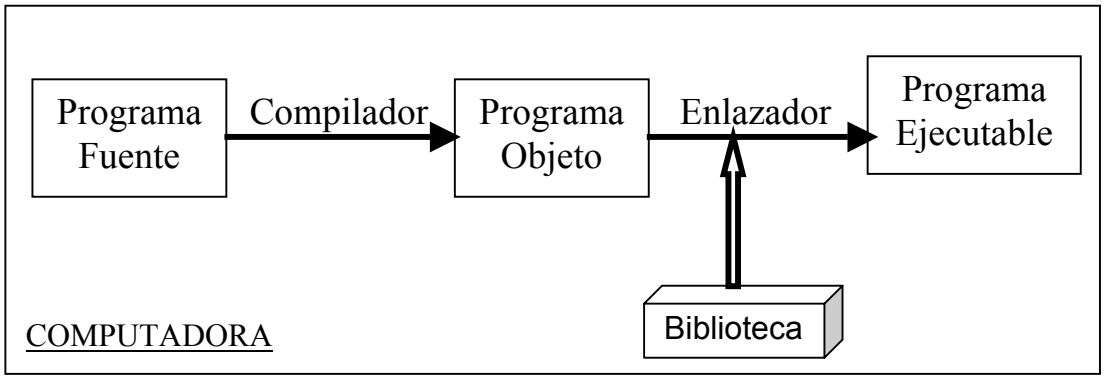


Los *intérpretes* realizan la traducción sentencia por sentencia y garantizan que cada sentencia traducida a código objeto sea ejecutada al momento. Una vez que una sentencia ha sido ejecutada se traduce y ejecuta la siguiente sentencia que aparezca en el código fuente. Los *compiladores* traducen todo el código fuente generando el código objeto del programa completo, el cual podrá ser ejecutado de una vez.

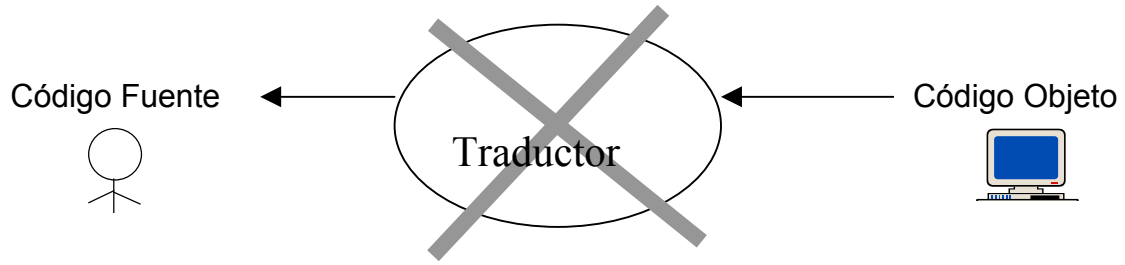


La compilación es un proceso de traducción más rápido que la interpretación y permite, a diferencia de ésta, guardar los documentos en código objeto para que puedan ser ejecutados nuevamente sin necesidad de realizar la traducción otra vez. Por otra parte, durante la traducción se encuentran los errores de sintaxis en el código fuente o sea, las sentencias mal escritas, si este proceso se realiza a través de un compilador el mismo se convierte además en un proceso de depuración de errores que culmina cuando el programador ha eliminado todos y cada uno de los errores de este

tipo, por lo que se puede asegurar que el código fuente es sintácticamente correcto. Los compiladores modernos detienen la traducción cuando aparece algún error sintáctico, permitiendo al programador corregirlo y continuar la traducción.
Los ambientes de desarrollo incluyen compiladores o intérpretes, por ejemplo, para todas las versiones de Pascal y de C se han construido compiladores. La mayoría de las versionesiniciales de Basic estaban montadas sobre intérpretes. Por las ventajas que ellos reportan, en la actualidad se construyen más compiladores que intérpretes, de ahí que la mayoría de los ambientes modernos incluyen un compilador.
Por lo general estos sistemas de programación permiten guardar tanto los códigos fuentes (archivos con extensión PAS, DFM, etc. en Pascal, con extensión C, H, etc. en lenguaje C, por sólo lo citar algunos ejemplos) como los códigos objetos (archivos con extensión DCU, OBJ, etc.). Los archivos con extensión EXE contienen código objeto previamente preparados (enlazados con las bibliotecas necesarias) para una total autonomía. Esa es la razón por la que al hacer doble click sobre ellos, se ejecuta directamente la aplicación en cuestión, sin necesidad de ejecutar la herramienta utilizada para su elaboración. Sin embargo, si se hace doble click sobre un archivo, por ejemplo, PAS se cargará automáticamente la versión de Pascal o Delphi instalada en la máquina y éste podrá ser ejecutado sólo desde allí. O sea, se puede concluir que un código sólo puede ser ejecutado en una máquina si es un código objeto o si se encuentra instalado en dicha máquina el ambiente de desarrollo que contiene al compilador o intérprete correspondiente.



Vale aclarar que el proceso inverso a la compilación no es posible de manera directa, por lo que los programadores son responsables de guardar los códigos fuentes de sus programas para futuras modificaciones. De no ser así, será necesario utilizar otras herramientas para dicha conversión.



2.- Elementos generales de la programación.

A continuación se darán algunos términos generales de la programación. Se analizarán primeramente los conceptos que encierran y posteriormente se pondrán ejemplos de ellos en un lenguaje de programación.

Vale aclarar que Object Pascal es una de las más recientes versiones de Pascal y que los contenidos que se abordan se mantienen invariantes desde el surgimiento de los primeros compiladores de Pascal. Por esta razón se utilizará de manera genérica Pascal para hacer referencia a cualquier versión de éste. Lo mismo sucede con el lenguaje C y cualquiera de sus versiones.

Identificadores

Los *identificadores* son cadenas alfanuméricas, o sea, formadas por letras y dígitos, que sirven para identificar, o sea nombrar, algún elemento de un programa. Estas cadenas deben comenzar siempre con letra y pueden contener el símbolo subrayado: "_". Se usan para denotar variables, constantes, módulos, etc.

En cada lenguaje e incluso versión de él pueden haber variantes en la cantidad de caracteres que se admita para formar dichas cadenas.

Es necesario aclarar que el Pascal y el C no permiten las letras acentuadas como parte de los identificadores. El primero no diferencia las minúsculas de las mayúsculas y el segundo sí.

Ejemplos válidos

1. A
2. Alfa
3. Radio1
4. CentroDeMasa
5. Perimetro_del_cuadrado

Ejemplos no válidos

1. 1Beta
2. Aα3
3. BaseDeTriángulo
4. Longitud-Cilindro

En los ejemplos no válidos, el primero comienza con un dígito, en lugar de con una letra. El segundo incluye un carácter no permisible para identificador, que es la letra griega alfa, α. El tercero es incorrecto, pues la segunda "a" está acentuada y el último, porque usa el signo menos, en lugar del signo de subrayar, para separar las palabras Longitud y Cilindro.

Los estándares de codificación actuales usan el estilo mostrado en el ejemplo válido no. 4, o sea, cada palabra que compone el identificador comenzando con mayúscula y el resto de las letras minúsculas.
Recomendaciones para el uso de identificadores.
1. Elegir identificadores que al leerse refieran el contenido del elemento en cuestión.
2. Elegir identificadores que sean pronunciables y entendibles.
3. No usar abreviaturas a menos que sean muy claras.
4. Evitar usar caracteres numéricos que puedan confundirse con letras como son los casos de 1(uno) y l (ele minúscula), 2 y Z, 0 y O.

Palabras reservadas

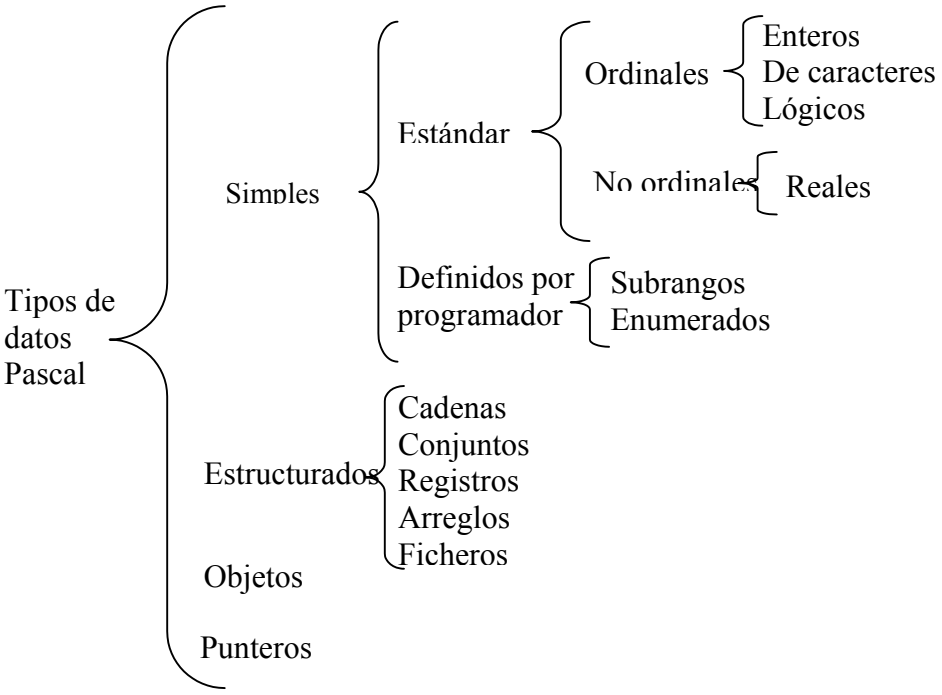
Las *palabras reservadas* son cadenas de caracteres alfanuméricos cuyo uso, como su nombre lo indica, está reservado por el lenguaje. Esto significa que no se pueden utilizar como identificadores de elementos en los programas.

Ejemplos Pascal
- **procedure**
- **type**
- **begin**
- **var**
- **const**
- **end**

Existen otras muchas palabras reservadas en el lenguaje. Para poder diferenciarlas de los demás identificadores en los documentos se usarán en negritas. Así se diferencian, generalmente en los editores de código.

Tipos de datos.
Un *tipo de dato* define un tipo de valor (numérico, de caracteres, lógico, etc.) y un rango de sus posibles valores, así como la cantidad de memoria que se necesita para almacenarlo. Los tipos de datos se utilizan para construir otros identificadores contenedores de datos.
Algunos ejemplos de tipos de datos son:
- el tipo definido por los valores numéricos enteros entre 0 y 255 y que necesitan sólo un byte de memoria para almacenarse,
- el tipo definido por todos los caracteres alfanuméricos incluyendo letras mayúsculas, minúsculas, dígitos y el resto de los caracteres posibles y que necesitan sólo un byte de memoria para almacenarse,
- el tipo definido por los valores lógicos verdadero y falso y que necesitan sólo un byte de memoria para almacenarse,
- el tipo definido por los valores numéricos enteros, con o sin signo, entre –2147483648 y 2147483647 y que necesitan 4 bytes de memoria para almacenarse,
- el tipo definido por los valores numéricos reales entre 5.0 x 10^–324 y 1.7 x 10^308 de 15 a 16 dígitos significativos y que necesitan 8 bytes de memoria para almacenarse.

Los tipos de datos del Pascal se clasifican de la siguiente manera:



Se dice que un tipo es ordinal, si sus valores responden a un orden dado, por lo que es posible hablar del predecesor y sucesor de cualquiera de ellos. Nótese que el concepto de ordinal no es inherente a los números reales.
Algunos de los tipos de datos simples estándar definidos en Object Pascal (Delphi versión 5.0) son:

- Enteros

Tipo	**Rango**	**Bytes**
Integer	–2147483648..2147483647	4
Cardinal	0..4294967295	4
Shortint	–128..127	1
Smallint	–32768..32767	2
Longint	–2147483648..2147483647	4
Int64	–2^63..2^63–1	8
Byte	0..255	1
Word	0..65535	2
Longword	0..4294967295	4

- Reales

Tipo	Rango	Dígitos significativos	Bytes
Real	2.9 x 10^–39 .. 1.7 x 10^38	11–12	6
Single	1.5 x 10^–45 .. 3.4 x 10^38	7–8	4
Double	5.0 x 10^–324 .. 1.7 x 10^308	15–16	8
Extended	3.6 x 10^–4951 .. 1.1 x 10^4932	19–20	10
Comp	–2^63+1 .. 2^63 –1	19–20	8
Currency	–922337203685477.5808.. 922337203685477.5807	19–20	8
Real*	5.0 x 10^–324 .. 1.7 x 10^308	15–16	8

* El tipo Real se mantiene por compatibilidad con versiones anteriores.

- Caracteres

Tipo	Rango	Bytes
Char	Todos los caracteres del código ASCII	1

Para este tipo el orden se define según la disposición de los caracteres en el código ASCII

- Lógicos

Tipo	Rango	Bytes
Boolean	**false, true**	**1**

Para este tipo el orden es: false precede a true.

- Cadenas de caracteres

Los tipos de cadenas de caracteres, a diferencia del char, permiten manipular cadenas de más de un carácter. La memoria que necesitan para ser almacenados depende, en general, de la cantidad de caracteres y cada tipo de cadena puede disponer de una cantidad de bytes máximos según el tipo en cuestión como puede verse a continuación.

Tipo	**Rango**	**Bytes**
ShortString	255 caracteres	2 a 256
AnsiString	~2 ^31 caracteres	4 bytes a 2GB
WideString	~2^30 caracteres	4 bytes a 2GB

En Object Pascal, el tipo String se mantiene por compatibilidad con versiones anteriores y tiene las mismas características del AnsiString.

Por otra parte, algunos lenguajes dan libertad a los programadores para que definan sus propios tipos de datos. Esto constituye una gran ventaja pues permite trabajar con tipos de datos más cercanos al problema concreto que se pretende resolver y por tanto aumenta la claridad de los programas.

La declaración de tipo de dato, en Pascal, se realiza mediante la palabra reservada **type** y sigue la siguiente sintaxis:

```
type
  identificador de tipo= tipo;
```

donde:
identificador de tipo: es el identificador que se asocia al tipo que se está declarando y a través del cual se hará referencia al mismo.
tipo: la especificación de los valores que pueden ser tratados a través del identificador, así como, la estructura que tendrán los mismos. Para la formalización del tipo, se utilizan otros tipos ya sean estándar o previamente declarados por el usuario.

Las declaraciones de tipo no se asocian con un espacio físico de memoria, o sea no consumen memoria, ya que estos no se utilizan para almacenar valores, sino para hacer explícito el tipo de los valores que almacenarán otros identificadores.
Durante la primera parte de este tema de programación sólo se trabajará con tipos de datos simples estándar, por lo que no será necesario hacer declaraciones de nuevos tipos de datos. En la medida que se avance en éste y en el resto de los cursos de programación que se imparten durante la carrera se estudiarán los restantes tipos de datos.

Constantes

En muchas situaciones se utilizan valores constantes para realizar diferentes cálculos. Tal es el caso, por ejemplo, de la constante Pi tan utilizada en las matemáticas para cálculos trigonométricos. Como su nombre lo indica, las *constantes* son valores que no varían en ningún cálculo.
Declarar o definir una constante en un programa significa declarar un identificador a través del cual se hará referencia a un valor constante en el programa, o sea un identificador que se asocia a un valor que no cambia durante la ejecución.
Los lenguajes de programación incluyen muchas constantes ya definidas, que el programador puede utilizar y generalmente brindan a éstos la posibilidad de declarar sus propias constantes.
La declaración de las constantes, en Pascal, se realiza a través de la palabra reservada **const** y responde a la sintaxis siguiente:

```
const
   identificador de constante[, identificador de constante...][: identificador de tipo]= valor;
```

donde:
identificador de constante: es el identificador, escogido por el programador para referirse a la constante en el cuerpo del código.
identificador de tipo: el identificador de tipo de la constante, es opcional.
valor: valor al cual hará referencia la constante en el cuerpo del código.

Nota: Al definir la sintaxis se usarán los corchetes para indicar algo que es opcional, o sea que no es obligatorio que aparezca dentro de la sentencia.

Ejemplos
const
max= 100;
ConstanteCalibracion= 2.956834544;

En el código donde aparezcan las declaraciones anteriores se puede hacer referencia a los valores 100 y 2.956834544 a través de los identificadores *max* y *ConstanteCalibracion* respectivamente. Esto reporta las siguientes ventajas, en primer lugar donde quiera que sea necesario utilizar esos valores se pueden colocar sus identificadores, en segundo lugar y como consecuencia de lo anterior, si fuese necesario en un momento posterior modificar algunos de estos valores, bastará con modificar la declaración de la constante en lugar de tener que modificar todos aquellos lugares del código donde aparezcan.

Variables

En los lenguajes de programación se le llama variable a todos los identificadores a través de los cuales se pueden almacenar datos en la memoria y acceder a ellos. Las variables pueden referenciar datos simples o estructuras de datos lo cual depende de la forma en que haya sido declarada la variable.
A cada variable se asocia una zona de memoria de un tamaño suficiente para almacenar sus valores.
La declaración de variable, en Pascal, se realiza mediante la palabra reservada **var** y sigue la siguiente sintaxis:
var
identificador de variable[, identificador de variable, ...]: identificador de tipo;
donde:
identificador de variable: es el nombre, escogido por el programador para hacer referencia, durante el cuerpo del código, al valor almacenado en la zona de memoria correspondiente a la variable.
identificador de tipo: identificador de tipo estándar o algún otro tipo previamente definido por el programador, que permite al compilador conocer la memoria que debe reservar para el uso de dicha variable y además hacer chequeos de compatibilidad durante la ejecución del código.

A diferencia de las constantes, las variables pueden tomar diferentes valores durante la ejecución del código. Cada vez que una variable cambia su valor se actualiza la zona de la memoria donde ella se almacena, por lo que el valor almacenado allí anteriormente se pierde.

Ejemplos
var
DiametroInterior, DiametroExterior: real;
Edad: byte;
CantidadTrabajadores: integer;
Consonante, Vocal: char;
EsVisible: boolean;
Parrafo: string;

En las variables *DiametroInterior*, *DiametroExterior* se pueden almacenar valores reales y ocuparán cada una 8 bytes, mientras que en *Edad* y *CantidadTrabajadores* se pueden almacenar números enteros. Sin embargo, los enteros que se almacenen en *Edad* sólo pueden estar en el rango entre 0 y 255 y ocuparán 1 byte de memoria, mientras que *CantidadTrabajadores* permitirá guardar valores enteros en el rango de –2147483648 a 2147483647 y consumirá 4 byte de memoria para ello. Las variables no numéricas *Consonante* y *Vocal* permitirán manipular a través de ellas caracteres (uno en cada instante de tiempo) y para ello consumirán 1 byte de memoria. *EsVisible* podrá guardar, en 1 byte, uno de los valores true o false. El último ejemplo muestra la variable *Parrafo*, a través de la cual se puede almacenar una cadena de caracteres y ocupará desde 4 bytes hasta 2GB según la cantidad de caracteres que contenga la cadena. La cantidad de bytes real que ocupe se puede calcular como 1 + 1 byte por cada carácter de la cadena.
Si en algún momento de la ejecución del código se trata de almacenar en una de estas variables un valor no permitido el compilador generará un mensaje de error.

Declaraciones

En todos los programas existen distintas zonas en las que es posible hacer declaraciones de tipo, constantes y variables. Cada declaración puede ir precedida de la palabra reservada **type**, **const** o **var** según sea el caso, pero se recomienda para mayor claridad agrupar todas las declaraciones de un tipo de identificador juntas. Por ejemplo:

```
const
  max= 100;
  ConstanteCalibracion= 2.956834544;
var
  DaimetroInterior, DiametroExterior: real;
  Edad: byte;
  CantidadTrabajadores: integer;
  Consonante, Vocal: char;
  EsVisible: boolean;
  Parrafo: string;
```

El orden en que se realicen las declaraciones no es significativo, pero debe respetarse siempre la **regla de precedencia de la declaración** que establece que "**solo pueden usarse identificadores previamente declarados**". En otras palabras, esto significa que para usar algún identificador es necesario que el mismo haya sido declarado anteriormente, de lo contrario el compilador reportará el error: "Identificador desconocido".
La zona de declaraciones permite que al inicio de la ejecución de un programa, el compilador reserve memoria para todas las constantes y variables según el tipo de cada una, no así para los tipos como se explicó anteriormente.

Nota: durante el curso se utilizará como parte del estándar de codificación el convenio de colocar una T mayúscula como prefijo a los tipos de datos definidos por el programador.

3.- Expresiones.

Expresión: conjunto de operandos combinados entre sí, a través de operadores. Los operadores indican la operación a realizar entre los operandos involucrados.
A toda expresión se asocia un valor que depende del valor de cada operando en el momento de su evaluación y de las operaciones indicadas entre ellos por los operadores.

Los operadores, según la cantidad de operandos que combinan, pueden ser:

- Unarios, operan sobre un solo operando. Tal es el caso del cambio de signo en los números y de la potenciación.
- Binarios, operan sobre dos operandos. Ejemplo el operador de suma, el de multiplicación, etc.

Y según el resultado de la operación en:

- Aritméticos: operan sobre operandos numéricos.
  Suma: +
  Diferencia: –
  Multiplicación: *
  División: /
  División entera: div
  Resto de la división: mod
  Cambio de signo: –
  Si los operandos son del mismo tipo, los operadores +, –, * dan como resultado valores del mismo tipo de los operandos. Si alguno de ellos es real, el resultado será un real del mismo tipo, aun cuando el otro operando sea de algún tipo entero.
- Lógicos
  - ✓ Relacionales: operan sobre cualquier par de operandos que sean del mismo tipo.
    Igual que: =
    Menor que: <
    Menor o igual que: <=
    Mayor que: >
    Mayor o igual que: >=
    Diferente de: <>
  - ✓ Conectivas lógicas: operan sobre operandos lógicos.

Negación lógica: *not*

Conjunción lógica: *and*

Disyunción lógica: or
Disyunción exclusiva: xor (verdadero sí sólo uno de los operandos es verdadero)

Vale aclarar que existen otros operadores que no aparecen en la relación anterior pues se utilizan entre operandos de tipos que no se verán por ahora, como son: in, is, as, @, etc.
Los operandos de una expresión pueden ser constantes, variables y expresiones. Los identificadores de tipos de datos no pueden formar parte de una expresión, pues no son contenedores de datos.

Las expresiones según el resultado de su evaluación pueden ser:
Numéricas: al evaluarlas se obtiene un valor numérico producto de

la combinación de operandos numéricos a través de operadores aritméticos.
Lógicas: al evaluarlas se obtiene un valor lógico, producto de la combinación de operandos del mismo tipo a través de operadores de relación, o de otras expresiones combinadas a través de conectivas lógicas.
Precedencia de los operadores:

Operadores	**Prioridad**
not	0
*, /, div, mod, and,	1
+, -, or, xor	2
=, <>, <, >, <=, >=	3

Para violar la precedencia de los operadores, se pueden utilizar paréntesis dentro de las expresiones.
Al evaluar una expresión se realizan las operaciones de izquierda a derecha, y según la precedencia de los operadores que aparecen en ella.

Ejemplos de expresiones
1. x + y * z
2. Alfa * 22 – h
3. Pi * Radio * Radio
4. (h + s) / (n * q)
5. EsVisible **and** (a > b)
6. **not** (x = z) **or** (b < 10)

Nótese que en el último ejemplo, primero se realiza la negación de la expresión x = z y después se realiza la disyunción entre el resultado de la negación anterior y la expresión b<10, pues **not** es un operador de mayor precedencia que **or**. Si por el contrario, se deseara la negación de la disyunción entonces sería necesario usar paréntesis para violar dicha precedencia **not(**(x=z) **or** (b<10)).

4.- Asignación.

La sentencia de asignación es aquella a través de la cual se asigna un valor a una variable y responde a la siguiente sintaxis:

identificador de variable:= expresión;

Esta instrucción asigna a la variable, especificada en la parte izquierda, el valor resultante de la evaluación de la expresión dada. O sea, primero se evalúa la expresión y luego el resultado se deposita en la localización de memoria asociada a la variable en cuestión. El

sentido de la asignación siempre es de derecha a izquierda, por lo que la variable que debe recibir el valor debe encontrarse siempre a la izquierda de la asignación, que se representa por ":=". Es importante resaltar que una asignación no es una igualdad, aunque a veces por comodidad se lea de esa forma.
Es necesario aclarar que el tipo de la variable debe ser compatible al del resultado de la evaluación de la expresión. De lo contrario el compilador generará un error de incompatibilidad de tipos.

Ejemplos:
1. x:= x + 1;
2. AreaCirculo := Pi * Radio* Radio;
3. AreaTriangulo := Base * Altura;
4. EsVisible:= not EsVisible;
5. Resultado:= x * 2 >= y;
6. Resto:= k mod n;
7. AreaSemiCirculo:= AreaCirculo / 2;

Todos los ejemplos anteriores requieren, por supuesto, que los identificadores *x*, *AreaCirculo*, *Radio*, *AreaTriangulo*, *Base*, *Altura*, *EsVisible*, *Resultado*, *y*, *Resto*, *k*, *n* y *AreaSemiCirculo* hayan sido previamente declarados.
Algunas reglas de compatibilidad de tipos para la asignación:
- A variables reales se les puede asignar valores enteros, lo contrario no es posible.
- A variables de algún tipo de los enteros/reales se les puede asignar valores enteros/reales de otro tipo, siempre y cuando sean de un rango contenido en el tipo de la variable destino. Por ejemplo, si Cantidad es de tipo integer y n de tipo byte, la asignación Cantidad:= n+2, es posible, lo contrario no es posible.

5.- Comentarios

Para aumentar la claridad del código es una buena práctica documentarlo con comentarios. Un comentario no es más que algún texto que se coloca dentro del código para hacer éste más explícito y que es ignorado por el compilador durante la ejecución.
En Object Pascal los comentarios se pueden escribir de dos formas distintas:
a) Un texto comenzando con doble slash "//". El compilador buscará el final del comentario donde encuentre un fin de línea (enter)
b) Un texto entre llaves "{}"

6.- Separadores

Todo lenguaje de programación establece una forma de indicar o reconocer el fin de una sentencia. En muchos se utiliza el *Enter* o fin de línea como separador, lo cual tiene el inconveniente de que obliga a escribir una sentencia por línea.
En Pascal se utiliza como separador entre sentencias el punto y coma ";" y esto da la posibilidad de escribir más de una sentencia en una línea de código.

Tema V: Estructuras básicas de la programación

Al inicio de la década del 60 comenzaron a aparecer trabajos sobre la programación estructurada. De manera general, todos de una forma u otra coincidían en afirmar que para escribir códigos claros y correctos sólo se necesitaban tres estructuras de programación:

a) secuencial
b) alternativa
c) repetitiva

por lo que debían evitarse los saltos incondicionales (a menos que se requiera para simular alguna estructura básica no implementada en el lenguaje), Exit, Halt, etc.

Esto ha trascendido hasta nuestros días, por lo que durante los cursos de programación se estudiarán diferentes algoritmos que serán implementados usando solamente estas tres estructuras.

La *estructura secuencial* es simplemente la idea de que las sentencias de un código son ejecutadas una tras otra en el orden en que aparecen en él. Más que una estructura propiamente dicha es una abstracción para la organización. A través de estructuras secuenciales se implementan fragmentos lineales de los algoritmos.

1.- Programación Modular. Tipos de módulos.

La investigación de la modularidad en informática procede del principio enunciado por Descartes en su obra "El Discurso del Método": "Dividir cada una de las dificultades que examinaré en tantas partes como se pueda y se requiera para resolverlo mejor..."

Para construir los programas, aplicando este principio, se procede por subdivisiones sucesivas descomponiendo los tratamientos en diferentes niveles, partiendo siempre del más global posible hasta llegar a los tratamientos elementales.

El objetivo fundamental de la modularidad es el de descomponer los procesos complejos en una sucesión de procesos más simples. El efecto es la simplificación del trabajo del analista o del programador, de permitir una verificación rápida de la coherencia lógica del conjunto y de permitir finalmente un mantenimiento más fácil de los sistemas.

Para ejemplificar este proceder analícese el siguiente ejemplo. Sea una empresa en la fecha en que debe calcularse la nómina para el pago a los trabajadores. Esto requiere que para cada trabajador se realicen las siguientes operaciones, entre otras:

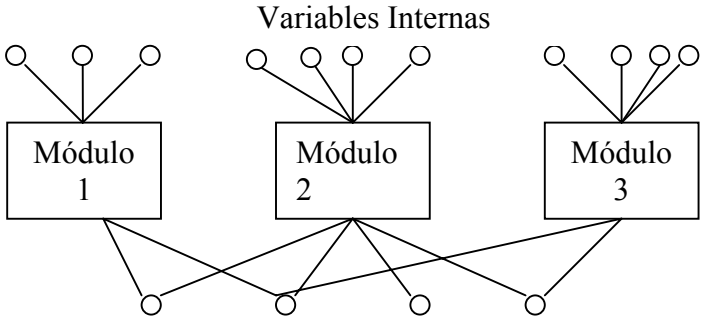
1ro. Calcular el salario bruto
2do.Calcular los descuentos
3ro. Calcular el salario neto

Cada uno de estos pasos constituye una unidad elemental de tratamiento, pudiendo ser a su vez subdivididas en órdenes cada vez más elementales. Así el Cálculo del salario bruto puede subdividirse en:

1ro. Obtener el número de horas trabajadas
2do. Multiplicar el número de horas trabajadas por la tasa por horas de la plaza ocupada.
3ro. Calcular las horas suplementarias, etc.

Nótese que se ha partido de un problema global que ha sido subdividido en tratamientos lógicos más sencillos, los cuales a su vez han vuelto a ser divididos en tratamientos lógicos más sencillos.

De una manera más formal se puede definir la concepción modular como la técnica que consiste en dividir la solución de un problema en unidades lógicas independientes llamadas módulos, de tal forma que cada uno de estos módulos pueda ser programado, compilado y probado independientemente de los otros.
En programación modular se pueden distinguir dos tipos de datos. Primeramente están aquellos que son usados por un módulo solamente y que son llamados datos internos o más exactamente variables internas. En segundo lugar están aquellos que son usados por más de un módulo y que son llamados externos o más exactamente variables externas.



Dicho de otra manera un módulo se comunica con otro mediante variables externas; las variables internas son simplemente para el almacenamiento y cálculo dentro del módulo.
Es conveniente distinguir entre las diferentes formas en que un módulo particular puede utilizar una variable externa determinada. Ellas son:

a) Para acceder a su valor

El módulo sólo se refiere a la variable, por ejemplo toma decisiones en función del contenido de la variable, o suma su contenido a otra variable, etc. Pero no puede de ninguna manera, modificar su contenido. Por ejemplo, para calcular la ganancia obtenida por la venta de un producto se obtiene la diferencia entre el precio unitario del producto y su costo unitario. Un módulo que lleve a cabo tal operación necesitará hacer referencia al precio y costo unitarios del producto, pero nunca modificarlos.

b) Para modificar su valor

En este caso el módulo simplemente modifica la variable, el contenido previo de la variable no tiene efecto a la salida del módulo. Por ejemplo, cuando se hace una reservación para un vuelo de avión, el estado de un asiento cambia pasando a ocupado. El módulo que realiza semejante tratamiento cambiará el valor de la variable EstaOcupado a true, perdiéndose el valor previo de dicha variable.

c) Para ambas cosas

El módulo toma el valor de la variable y la modifica. Por ejemplo, un módulo donde se actualiza el salario de un trabajador por concepto de antigüedad o cambio de tridente. En este caso al salario actual del trabajador se le suma una cantidad en dependencia del caso concreto. Para ello por supuesto se hace referencia al salario actual para, a partir de él, calcular el nuevo salario.

Algunos criterios para un buen diseño de módulos

- Debe existir la menor cantidad de variables externas.

- La duplicación de instrucciones en diferentes partes debe evitarse siempre que sea posible, agrupándolas en módulos.
- Aquellos fragmentos de código que puedan ser requeridos en otros programas deben ser aislados en módulos separados.
- Todas las partes que sean susceptibles a cambios en un futuro deben ser aisladas en módulos separados.
- Los módulos no deben ser ni muy triviales ni muy complejos.
- Cada módulo debe ser una parte lógica de un programa concreto, con una funcionalidad clara y concreta.

Estos criterios son, en su mayoría poco objetivos, y el éxito con que sean llevados a la práctica dependen de la experiencia, creatividad y habilidad del programador. Por supuesto al tratar de diseñar siguiendo algunos de estos criterios seguramente se encontrarán conflictos, a los que los programadores deben dar solución según las características del caso en cuestión. Una restricción importante del diseño modular, que se debe tener presente al solucionar conflictos de diseño, es que cada módulo debe tener un punto de entrada y un punto de salida solamente.

Algunas ventajas de la modularidad

- Los códigos son menos vulnerables a cambios.
- La vida activa del código se extiende.
- Uso más efectivo de los recursos.
- Facilita las pruebas que se realizan al código.
- Al programar módulos generales, disminuye la posibilidad de errores en el código. Mientras más se usa un módulo más depurado estará su código.
- Aumento de la productividad de los programadores.
- Permite abordar la codificación en equipos.
- Facilita el control de los programadores y la distribución de los recursos.

Tipos de módulos

Los módulos pueden clasificarse de diferentes maneras:

a) Atendiendo a la función que realizan dentro de las aplicaciones
   - Director: es el responsable del flujo de control hacia los distintos módulos.
   - De tratamiento: son los que implementan los segmentos lógicos, o sea los tratamientos elementales. Cada uno desempeña en su totalidad, una función claramente definida. Es una entidad que incluye una zona reservada para su uso exclusivo. Se caracterizan por devolver el control únicamente al módulo que se lo entregó. Deben concebirse lo más general posible, para que puedan ser portados a diferentes aplicaciones.
   - De uso múltiple: agrupan un conjunto lógico de sentencias comunes a varios módulos. Por lo general son útiles en varios programas, por lo que también se les llama utilitarios.

- De entrada/salida: realizan las funciones especializadas de entrada/salida. Son individualizados en las aplicaciones, por su falta de generalidad. Incluyen las operaciones de apertura, acceso y cierre de los ficheros. Generalmente están vinculados a módulos de uso múltiples para las validaciones, se sirven de ellos.
- De atención a un mensaje (manipulador de evento): aquel que responde directamente a un mensaje enviado a un objeto. Cuando ocurren eventos durante la ejecución de la aplicación se envían mensajes a quienes deben atender a dichos eventos. Estos módulos contienen las respuestas necesarias.

b) Atendiendo a su implementación
   - Programa: módulo director que implementa las decisiones que dirigen el flujo de los datos a los módulos correspondientes. Están compuestos por subprogramas y pueden hacer uso de bibliotecas disponibles.
   - Subprograma: módulo autónomo relativamente independiente que realiza una operación o tarea específica. Se define una vez y puede ser utilizado (invocado o llamado) cada vez que se necesite realizar la tarea para la que fue diseñado. Es compilado cuando se compila el módulo dentro del cual está contenido. Un subprograma puede ser un módulo de tratamiento, de entrada/salida de uso múltiple o, incluso, de atención a un mensaje. En la literatura con frecuencia se les refiere también como rutinas o subrutinas.
   - Biblioteca: módulo que agrupa un conjunto de subprogramas independientes, pero que guardan alguna relación entre sí, de acuerdo a la tarea que desempeña cada uno. Se ponen a punto y se compilan por separado y se usan en las distintas aplicaciones. Por lo general agrupan módulos de uso múltiple.

Estructura general de un módulo.

Todo módulo en Pascal, independientemente de su tipo, tiene la siguiente estructura:

Encabezamiento
Zona de declaraciones internas
Cuerpo

El *Encabezamiento* contiene la especificación del tipo de módulo y el identificador del mismo.

La Zona de declaraciones internas contiene las declaraciones de tipos, constantes y variables internas o locales al módulo, así como, la declaración de otros módulos internos a éste.

Por último aparece el *Cuerpo* del módulo, que contiene todas las sentencias ejecutables del módulo encerradas entre las palabras reservadas **begin** y **end** que hacen las veces de paréntesis abierto y paréntesis cerrado respectivamente.

Cuando un módulo recibe el control, comienza la ejecución del mismo por la primera sentencia que aparezca en el cuerpo.

En Pascal existen tres tipos de módulos:

- Programas
- Subprogramas: procedimientos y funciones
- Units

Los programas se encabezan con la palabra reservada **programa** y terminan en punto "." después del **end** final.

Los ambientes modernos, por lo general incluyen automáticamente el programa una vez que se inicia la construcción de la aplicación y su cuerpo contiene una serie de sentencias o invocaciones que debe ejecutar cualquier aplicación siempre que se inicie. Rara vez es necesario modificar el cuerpo del programa.
Este es el caso del ambiente Delphi, cuando se inicia la creación de un nuevo proyecto, ya este tiene incluido el programa, cuyo cuerpo será lo primero que se ejecute cuando comience la ejecución de la aplicación. El cuerpo del programa contiene las instrucciones necesarias para que la aplicación se inicie y quede en espera de la ocurrencia de algún evento. El programador, entonces, sólo requiere programar los manipuladores de cada evento y es muy poco probable que necesite modificar el cuerpo del programa, por lo que no se dedicará esfuerzo a desarrollar esta habilidad en el curso.
En Pascal los módulos de atención a mensajes son subprogramas, o sea son también Procedimientos y/o funciones.

Procedimientos y funciones. Transferencia de información.

En Pascal, y en la mayoría de los lenguajes de programación modernos, existen dos tipos de subprogramas, los Procedimientos y las Funciones, que se diferencian entre sí por la forma en que se declaran, se invocan y devuelven los resultados.
En cualquiera de los dos casos se distinguen dos momentos importantes:
1. Declaración: en este momento se define el identificador, y se hace explícito todo lo que es capaz de hacer el módulo, o sea se escribe su código completo.
2. Invocación: es el momento en el que desde algún otro lugar de la aplicación se ordena ejecutar el código correspondiente al módulo.

Procedimiento

3. Sintaxis de la declaración del procedimiento:

```
procedure identificador del procedimiento[(lista de parámetros formales)];
    [zona de declaraciones]
begin
    Cuerpo
end;
```

Acerca de la declaración:
- Se utiliza la palabra reservada **procedure** en su encabezamiento para indicar que se trata de un procedimiento.
- La lista de parámetros formales que se escribe opcionalmente a continuación del identificador, constituye la interfaz de comunicación del procedimiento con el exterior.
  La sintaxis de la lista de parámetros formales se detallará posteriormente.
- Después del encabezamiento aparece la zona de declaraciones en la que se definen, si existen, las variables, constantes, tipos de datos y otros subprogramas internos al procedimiento.
- Los elementos declarados en dicha zona son considerados locales al procedimiento, o sea, no podrá accederse a ellos desde otro lugar externo al procedimiento.

No es obligatorio que un procedimiento contenga declaraciones locales.

- Se definen subprogramas internos o locales a otro, cuando es posible desglosar la tarea que se realiza en otras tareas lógicas que pueden abordarse con relativa independencia, pero que sólo serán necesarias dentro de dicho módulo. Un módulo interno a un subprograma, nunca podrá ser invocado desde el exterior.

  Para el ejemplo del volumen del cilindro hueco, podría pensarse en un módulo interno que calcule el área de la base.

  Los subprogramas internos pueden ser de cualquier tipo.
- Las instrucciones a ejecutar en la solución del problema para el que fue diseñado el procedimiento constituyen el cuerpo del procedimiento y se escriben entre las palabras reservadas **begin** y **end**.
- Después del **end** que indica el fin del procedimiento se escribe un separador, o sea un punto y coma ";".

- Sintaxis de la invocación al procedimiento:

identificador del procedimiento(lista de parámetros actuales);

Como se ve, la llamada o invocación al procedimiento se realiza a través del identificador del procedimiento y los parámetros, si los tiene. Esta invocación constituye una instrucción más del programa y causa una transferencia del control de la ejecución al punto donde comienza el subprograma. Una vez que se haya completado la ejecución del procedimiento, el flujo del control regresará a la sentencia posterior a la invocación.

Ejemplos de encabezamientos de procedimiento

1. **procedure** MuestraGrafica;
2. **procedure** CaptarDatos(var RI, RE, Longitud: real);

Ejemplos de llamadas a procedimientos

1. CaptarDatos(RadioInterior, RadioExterior, Longitud);
2. MuestraGrafica;

Para que se entienda mejor el proceder del compilador ante una llamada a un procedimiento, véase el siguiente esquema donde los números indican el orden de ejecución de las sentencias.

```
Procedure MostrarCilindro;
    procedure CaptarDatos(var RI, RE, Longitud: real);
       begin
        //este es el cuerpo de CaptarDatos
        ... (4)
        ... (5)
        ... (6)
      end;
    ...
    //otros subprogramas internos a MostrarCilindro
    ...
begin
//este es el cuerpo de MostrarCilindro
... (1)
... (2)
CaptarDatos(RadioInterior, RadioExterior, Longitud);(3)
... (7)
...
end; //aquí termina MostrarCilindro
```

El esquema anterior muestra un procedimiento MostrarCilindro que contiene otro procedimiento llamado CaptarDatos (a esto se le llama procedimientos anidados). Dentro del cuerpo de MostrarCilindro se invoca a CaptarDatos, lo cual provoca que en ese momento de la ejecución se salte a ejecutar dicho procedimiento y cuando culmine la ejecución del mismo se continúe ejecutando el cuerpo de MostrarCilindro en la sentencia que le sigue a la invocación. Los números colocados a continuación de las supuestas sentencias intentan explicar el orden de ejecución.

Funciones

Son subprogramas que llevan a cabo una tarea concreta que siempre tendrá al menos un resultado que devolver. El resultado principal de una función se entrega a través de su identificador una vez que ésta se ejecuta. Por tanto, al ejecutar una función su identificador se comporta como un contenedor de datos, o sea como una variable. Por esta razón las funciones se invocan dentro de expresiones.

Sintaxis de la declaración de la función:

```
function identificador de la función[(lista de parámetros formales)]: tipo;
    [zona de declaraciones]
begin
    Cuerpo //debe contener siempre una sentencia del tipo identificador de la
                //función:= expresión o la equivalente result:= expresión
end;
```

Acerca de la declaración:

- Se utiliza la palabra reservada **function** en su encabezamiento para indicar que se trata de una función.
- Los parámetros formales que se escriben opcionalmente a continuación del identificador, constituyen los argumentos de la función.
- Después del encabezamiento aparece la zona de declaraciones en la que se definen, si existen, las variables, constantes, tipos de datos y otros subprogramas internos a la función.
- Los elementos declarados en dicha zona son considerados locales a la función, o sea, no podrán ser accedidos desde otro lugar externo a la función.
- No es obligatorio que una función contenga declaraciones locales.
- Al igual que para los procedimientos, una función puede tener otros subprogramas internos, que pueden ser procedimientos y/o funciones.
- Las instrucciones a ejecutar en la solución del problema para el que fue diseñada la función constituyen el cuerpo de la misma y se escriben entre las palabras reservadas **begin** y **end**.
- Después del **end** que indica el fin de la función se escribe un separador, o sea un punto y coma ";".
- Dentro del cuerpo de la función es obligatorio asignar algún valor al identificador de la función, o a una variable *result*, del mismo tipo de la función, que se crea automáticamente cuando se comienza a ejecutar la misma. Este valor será el resultado principal de la función.

Sintaxis de la invocación a la función:

Dentro de una expresión  ...identificador de la función(parámetros actuales)...

Como se ve, la llamada o invocación a una función se realiza a través de su identificador y los parámetros actuales. La invocación debe aparecer SIEMPRE dentro de una expresión.
Cuando el compilador evalúa una expresión, si encuentra el identificador de una función, transfiere del control de la ejecución al punto donde comienza la misma. Una vez que se haya culminado la ejecución de dicha función, el flujo del control regresará a completar la evaluación de la expresión que contiene la invocación, tomando para ello el resultado principal devuelto por la función como el valor necesario para completar la evaluación.

Ejemplos de encabezamientos de funciones

1. **function** Volumen(aRI, aRE, aLong: real): real;
2. **function** AreaCirculo(aR: real): real;

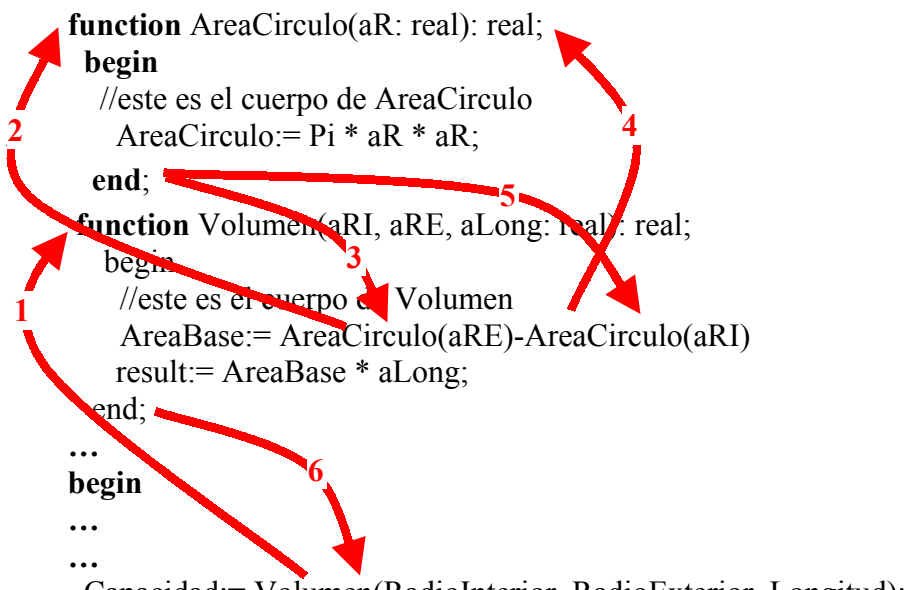
Ejemplos de llamadas a funciones

1. Capacidad:= Volumen(RadioInterior, RadioExterior, Longitud);
2. result:= (AreaCirculo(RadioExterior)-AreaCirculo(RadioInterior)) * Longitud;

Para que se entienda mejor el proceder del compilador ante una llamada a una función, véase para el fragmento que sigue el esquema de más abajo que muestra el orden de ejecución de las invocaciones.

```
…
function AreaCirculo(aR: real): real;
  begin
    //este es el cuerpo de AreaCirculo
   AreaCirculo:= Pi * aR * aR;
  end;
 function Volumen(aRI, aRE, aLong: real): real;
    begin
      //este es el cuerpo de Volumen
      AreaBase:= AreaCirculo(aRE)-AreaCirculo(aRI)
      Volumen:= AreaBase * aLong;
   end;
…
begin
…
…
Capacidad:= Volumen(RadioInterior, RadioExterior, Longitud);
...
...
end;
```

Esquema de la ejecución del fragmento anterior



El esquema anterior muestra dos funciones, *AreaCirculo* y *Volumen*, contenidas dentro de algún otro módulo que las invoca. Primeramente se trata de evaluar la expresión *Volumen* y asignar su resultado en la variable *Capacidad*, ello hace que el compilador se remita al código de la función *Volumen* para poder conocer el valor de este identificador, que no está almacenado en ningún lugar de la memoria, sino que se calcula en el momento que se necesita. Una vez dentro del código de *Volumen* se necesita conocer el valor de *AreaCirculo*, para lo cual se va a ejecutar su código con el argumento *aRE*. Cuando se ha completado la ejecución de *AreaCirculo* con el argumento *aRE*, se retorna su valor al punto donde fue invocada y continua la evaluación de la expresión. Para ello es necesario volver a calcular el área de un círculo ahora con el argumento *aRI* y por tanto se vuelve a ejecutar el código de la función *AreaCirculo*. El nuevo valor retornado por la función se utiliza para completar la evaluación de la expresión.
Después que se asigna un valor a AreaBase, se continúa la secuencia asignado a *result* el resultado de la expresión, que será el valor retornado por la función *Volumen*. Finalmente este resultado se asigna a la variable *Capacidad*.
Nótese que en ambas funciones se garantiza que al retornar se devuelva un valor del tipo de la función. Sin embargo para ello, en *AreaCirculo* se le asignó el resultado al identificador *AreaCirculo* mientras que en Volumen se le asignó a la variable implícita *result*. Ambos modos son correctos, pero es aconsejable el segundo estilo siempre que el lenguaje lo permita, pues se obtiene un código más uniforme y se evitan llamados recursivos, algo que se verá en próximos cursos de programación.

Cuadro resumen: Diferencias entre Función y Procedimiento.

	Procedimiento	Función
Encabezamiento	**Procedure**	**function**
Declaración	En el cuerpo el identificador no recibe valor nunca.	En el cuerpo el identificador (o result) recibe valor siempre.
Invocación	Una sentencia	Dentro de alguna expresión.

Transferencia de información

Los parámetros de los subprogramas constituyen la interfaz de comunicación de estos con el exterior y viceversa. A través de los parámetros se le envía información a los subprogramas y se recibe información de ellos. En el caso particular de las funciones el identificador también brinda información al exterior. Pero cómo ocurre esa transferencia de información hacia y desde los subprogramas.
La forma en que ocurre la transferencia de información está muy vinculada con la relación que guarda el módulo con los datos o variables externas, o sea, si es para acceder a su valor, para modificarlo o para ambas cosas. En dependencia de la relación existente los parámetros pueden ser:

- De entrada al módulo: a través de los parámetros de entrada el subprograma recibe datos que serán referenciados dentro de él. O sea, parámetros que serán usados en el procesamiento interno, pero no serán modificados. En este caso la comunicación es solamente hacia el subprograma.

- De salida del módulo: a través de los parámetros de salida un módulo devuelve resultados obtenidos producto de su procesamiento. En este caso la comunicación es desde el interior hacia el exterior del subprograma.
- De entrada y salida: a través de ellos un subprograma recibe una información desde el exterior que es modificada dentro de su procesamiento interno, devolviendo al exterior el último valor recibido. En este caso ocurre una transferencia en ambos sentidos, hacia y desde el subprograma.

En Pascal para la transferencia de información se han implementado parámetros de entrada o también llamados por valor y parámetros de entrada/salida o también llamados por referencia. Los primeros solamente permiten la transferencia hacia el subprograma, mientras que los segundos permiten la transferencia en ambos sentidos, hacia y desde el subprograma.
Se conoce como parámetros formales a los que se declaran en el encabezamiento del subprograma y que definen la interfaz de comunicación del mismo. Los parámetros actuales son los que se colocan en las invocaciones y por tanto constituyen la fuente de información o destino de esta.

Sintaxis de la declaración de los parámetros formales

([**var**] identificador[, identificador,...]: tipo[;[**var**] identificador[, identificador,...];...])

O sea, se escriben listas de identificadores separadas por punto y coma. Cada lista puede contener uno o más identificadores separados por coma y finaliza con la declaración del tipo de los identificadores que la conforman. La palabra reservada **var** se utiliza cuando el parámetro es de entrada/salida. Por tanto, ante la ausencia de la palabra **var**, se asume que el parámetro es de entrada solamente.
Ejemplos

1. **function** AreaCirculo(aR: real): real;
2. **procedure** CaptarDatos(var RI, RE, Longitud: real);
3. **function** ExisteElemento(aElemento: **string**; var Posicion: byte): boolean;

El ejemplo 1 muestra una función que tiene un parámetro de entrada y no tiene parámetros de entada/salida, pues su único resultado se devuelve en el identificador. El ejemplo 2, por su parte, es un procedimiento que sólo tiene tres parámetros reales que son de entrada/salida. Por último, el ejemplo 3 es una función booleana que recibe una información a la entrada por el parámetro aElemento y que puede recibir y/o devolver un valor entero de un byte por el parámetro de entrada/salida Posicion.
Vale aclarar que los tipos utilizados en una declaración de subprograma, deben ser tipos estándar o previamente definidos. O sea, en este espacio no se pueden hacer declaraciones de tipo.

Sintaxis de los parámetros actuales

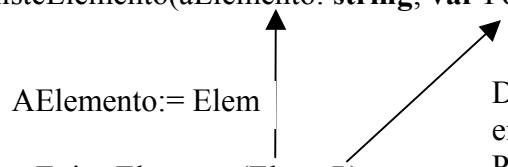
(identificador [, identificador,...])

O sea, una lista de identificadores de variables externas que contienen la información que se enviará al subprograma y donde se almacenarán los datos devueltos por éste.

Ejemplo
1. ... AreaCirculo(Radio)...
2. CaptarDatos(RadioInterior, RadioExterior, Longitud)
3. ... ExisteElemento(Elem, P)...

Como puede apreciarse los identificadores de los parámetros actuales no tienen que coincidir con los declarados en los parámetros formales, pero si **tienen que coincidir en cantidad, tipo y orden**. ¿Por qué?
Cuando se invoca a un subprograma para los parámetros de entrada ocurre una asignación del valor del parámetro actual al parámetro formal, de ahí que también se diga que esta es una transferencia por valor. Sin embargo, entre los parámetros de entrada/salida lo que realmente se transfiere es una dirección de memoria que debe ser compartida por el parámetro formal y el actual durante la ejecución del subprograma, lo cual garantiza que si durante la ejecución del subprograma se escribe algo en esa zona de memoria a la salida se pueda acceder a dicho información. Por esta razón a estos parámetros se les llama también por referencia. Esta situación se muestra en el siguiente esquema.

function ExisteElemento(aElemento: **string**; **var** Posicion: byte): boolean;



Dirección de memoria donde se encuentra el valor de P y a la cual Posición debe acceder para leer y escribir información

Nota: durante el curso se utilizará como parte del estándar de codificación el siguiente convenio. Colocar una a minúscula como prefijo a los parámetros de entrada.

Funciones y procedimientos suministrados más usados.

Todos los lenguajes de programación tienen incorporado un conjunto de funciones y procedimientos que los programadores pueden invocar sin tener que declararlos. Por ejemplo:

Funciones

- sin(x: real): real, devuelve el seno del argumento
- cos(x: real): real, devuelve el coseno del argumento
- abs(x: real): real, devuelve el valor absoluto del argumento
- sqr(x: real): real, devuelve el cuadrado del argumento

- sqrt(x: real): real, devuelve la raíz cuadrada del argumento
- ln(x: real): real, devuelve logaritmo natural del argumento
- exp(x: real): real, devuelve e^x
- strtofloat(s: string): real, convierte a un real el contenido de la cadena argumento
- strtoint(s: string): integer, convierte a un integer el contenido de la cadena argumento
- pred(x: algún tipo ordinal): el mismo tipo ordinal del argumento, devuelve el predecesor del argumento
- succ(x: algún tipo ordinal): el mismo tipo ordinal del argumento, devuelve el sucesor del argumento
- ord(elemento: algún tipo ordinal): byte, devuelve el ordinal del argumento
- chr(c: byte): byte, devuelve el código ASCII del argumento

Procedimientos
- inc(var i: integer, n: LongInt), incrementa i en n. Si no aparece n, entonces incrementa i en uno.
- dec(var i: integer, n: LongInt:), decrementa i en n. Si no aparece n, entonces decrementa i en uno.
- val(s: string; numero: integer; codigo: integer) o
  val(s: string; numero: real; código: integer), convierte una cadena a número, entero o real, si es posible
- str(numero: real; var s: string) convierte un número a cadena

Por supuesto la lista es bastante grande y no tiene sentido escribirla completa. El conocimiento de los procedimientos y funciones suministradas, se adquiere con la práctica, consultando los manuales de referencia de los lenguajes y sobre todo consultando la ayuda en línea del ambiente de programación.

Units. Criterios elementales de diseño de bibliotecas. (Materiales)

Una de las características más interesantes de las versiones de Pascal es la posibilidad de descomponer un programa grande en módulos más pequeños que se pueden compilar independientemente. Estos módulos se denominan *Unidades de Compilación Independientes* (*UNIT*)
Una *Unit* es una colección de declaraciones de constantes, tipos de datos, variables, procedimientos y funciones que pueden ser compartidas por varios programas. Se compilan y ponen a punto de manera independiente, obteniéndose una biblioteca que puede ser utilizada por otros programas, pero no se pueden ejecutar directamente.
Cada unidad es una biblioteca de declaraciones que se puede poner en uso en varios programas, y que permiten que éstos se puedan dividir y compilar de manera independiente.
Una vez que una unidad ha sido compilada y depurada no necesita compilarse más. El Pascal se encarga de enlazar la unidad al programa que la utiliza, empleando siempre menos tiempo que en la propia compilación.
Una unidad puede, a su vez, utilizar a otras unidades. Para que esto ocurra, o para que un programa Pascal se enlace con la unidad que necesita utilizar, hay que emplear luego del

encabezamiento de los mismos la cláusula **Uses** y a continuación el nombre o identificador de la UNIT a utilizar.

Ventajas de las UNITS

1. Facilita la independencia entre un grupo de programadores. Cada especialista puede concentrarse en la resolución de una parte del problema global, programando, probando y compilando el código en units,
2. Aumenta la reusabilidad por concepto de código portable a otras aplicaciones y por tanto, aumenta también la productividad de los programadores.
3. Disminuye el tiempo de compilación de la aplicación, pues una vez que las unit son compiladas no es necesario volver a compilarlas a menos que sufran modificaciones.

En Delphi se dispone de varias unidades estándar o suministradas, relacionadas con tareas específicas: generación de gráficos, gestión de pantalla, etc. Cualquier usuario puede, a su vez, crear sus propias units, enriqueciendo así los recursos básicos.

Sintaxis de la unit:
Las unidades tienen una estructura similar al resto de los módulos, como puede apreciarse a continuación.

Unit identificador de la unit;
Interface

[**Uses** lista de identificadores de otras unit;]

sección de interface

Implementation

[**Uses** lista de identificadores de otras unit;]

sección de implementación

[Begin
  zona de inicialización]
end.

Como puede apreciarse, una UNIT está constituida de tres partes: Interface, Implementation e Inicialization.

- El nombre de la unidad es un identificador que, por lo general coincidirá con el nombre del fichero físico que la contiene.
- La sección de **Interface** es la parte que sirve para conectar dicha unidad con otras unidades y programas. Esta sección se conoce como “la parte pública” de la unidad, ya que las restantes unidades y programas tienen acceso sólo a la información contenida en

esta sección. En ella se declaran los tipos de datos, constantes y variables, así como los encabezamientos de los procedimientos y funciones a los cuales tendrán acceso los usuarios de la unit. Sólo los subprogramas que aparecen en la sección Interface pueden ser invocados desde el exterior de la unidad.

- La sección **Implementation** es estrictamente de uso privado de la unit. Esta sección contiene el cuerpo de los todos los subprogramas cuyos encabezamientos aparecen declarados en la Interface (no es obligatorio repetir la lista de parámetros y sus tipos, pero es recomendable), así como aquellos de uso privado de la unidad. Las variables, constantes, tipos de datos o subprogramas declarados dentro de esta sección no pueden ser accedidos por cualquier otra unidad o programa.
- La zona de **Inicialization** puede o no aparecer. Ella contendrá instrucciones que sirven, por ejemplo, para inicializar variables, y hacerlas disponibles (a través de la Interface) a los módulos que las utilizan. La ejecución de estas instrucciones se efectúa antes de la ejecución de la primera sentencia ejecutable del cuerpo del programa que ponga en uso a la UNIT. Con frecuencia esta zona aparece vacía.
- La declaración de una UNIT termina en punto después del **"end."**.

Alcance de los identificadores

Los identificadores locales tienen mayor precedencia que los externos o globales, en otras palabras, si dentro de un módulo se declara un identificador igual a uno externo, dentro de dicho módulo tendrá validez el que haya sido declarado en él.

Dentro de un módulo cualquiera se puede acceder a los identificadores locales y a los globales a él, pero no a los identificadores locales a otros módulos que no lo contengan.
Por ejemplo:

```
Uses Unidad;
procedure Uno(P1: integer);
  var A: byte; C: real;
  function Dos(F1: real ): real;
     var A: real; B: integer;
     begin
     ....//A sería la variable local declarada dentro de la función
        //no se puede acceder a la variable A declarada dentro de Uno
        //se puede acceder a la variable A de la unit  utilizando la notación  identif. de la unit.
       //Identif. de la variable(Unidad.A)
       //se tiene acceso al parámetro P1 de Uno, a la variable C de Uno,
       //a la variable D de la unit y al parámetro F1 de Dos
      //se puede invocar cualquier subprograma público de la unit
      // no se puede invocar a Tres
     end;
begin
…....//A sería la variable local declarada dentro de Uno
       //no se puede acceder a las variables Ay B declaradas dentro de Dos
```

```
Unit Unidad;
Var A: char;
      D: rreal;
.....
end
```

```
      //se puede acceder a la variable A de la unit  por Unidad.A
      //se tiene acceso al parámetro P1 de Uno, a la variable C de Uno,
      //a la variable D de la unit
     //se puede invocar a Dos y a cualquier subprograma público de la unit
     // no se puede invocar a Tres
end;
procedure Tres(P1: char);
begin
…//se tiene acceso al  parámetro P1 de Tres y a todo lo público de la unit
  //se puede invocar a Uno, pero no a Dos.
end;
```

Cuando en una aplicación se trabaja con varias units puede ocurrir que en más de una de ellas se utilice un mismo identificador. En estos casos la notación para especificar el deseado en un momento dado es:
identificador de la unit**.**identificador deseado

Módulos de respuesta a mensajes.

Como se explicó anteriormente todo evento que ocurre durante la ejecución de una aplicación, provoca que se le envíe un mensaje a quien debe dar respuesta al mismo. Por ejemplo, hacer click sobre un botón, escribir un carácter en un edit, colocar el mouse sobre un control de la interfaz, escoger una opción de un menú, cerrar una aplicación, etc., son todos posibles eventos generados por los usuarios, a los cuales las aplicaciones deben dar una respuesta desempeñando un comportamiento preconcebido para ello. Los módulos de respuesta a mensajes, o como también se les dice de atención a eventos, contienen el código que implementa el comportamiento para responder a un mensaje.
En Pascal estos módulos son en términos de programación subprogramas, o sea procedimientos y funciones, pero que están asociados a los objetos que deben responder. Por ejemplo, una ventana con un botón, debe contener un módulo que responda al evento hacer click sobre el botón.
Cuando se codifica la respuesta a un evento que puede ocurrir durante la aplicación, se dice que se ha programado el manejador o manipulador de un evento. En Delphi cuando se desea programar un manejador de algún posible evento ocurrido en la interfaz, se selecciona el control sobre el cual se espera el evento y en el Object Inspector se busca el evento en cuestión y se hace doble click en él. Cuando el evento que se desea manejar es el click sobre el control, basta con hacer doble click sobre el control receptor del evento durante el diseño. De cualquier manera en el editor de código aparecerá el esqueleto (encabezamiento y estructura en general) de un módulo para que el programador escriba su cuerpo.
Los módulos de respuesta a mensajes provocados por eventos ocurridos en la interfaz, deben concebirse lo más desligados posible de los módulos de tratamiento para disminuir los impactos causados por futuros cambios y aumentar la reusabilidad del código.

Ejemplos.

A continuación se muestra un ejemplo de unit donde se ha agrupado un conjunto de operaciones auxiliares que pudieran ser necesarias en algunas aplicaciones.

```
unit auxiliar;

interface
function Potencia(aBase, aExponente): real;
function PromedioDeTresNumeros(aX1, aX2, aX3: real): real;
function PorcientoCantDeTotal(aCant, aTotal: real): real;
procedure IncrementaEnDos(var aI: real);
procedure NegacionDe(var aValor: boolean);
procedure IntercambiaReales(var aX, aY: real);
procedure IntercambiaEnteros(var aX, aY: integer);
procedure IntercambiaChar(var aX, aY: char);
procedure IntercambiaCadenas(var aX, aY: string);

implementation
{*}function Potencia(aBase, aExponente): real;
begin
  //si Y= X^a entonces ln Y = ln X^a
  //por propiedad del ln la expresión anterior es igual  ln Y =a ln X
  //aplicando exponencial a ambos lados e ^ln Y =e ^a ln X
  result:= exp(aExponente * ln(aBase));
end;
function PromedioDeTresNumeros(aX1, aX2, aX3: real): real;
begin
  result:= (aX1 + aX2 + aX3) / 3;
end;
function PorcientoCantDeTotal(aCant, aTotal: real): real;
begin
 result:= (aCant / aTotal ) * 100;
end;
{*}procedure IncrementaEnDos(var aI: real);
begin
 aI:= aI + 2;
end;
{*}procedure Niega(var aValor: boolean);
begin
 aValor:= not aValor;
end;
{*}procedure IntercambiaReales(var aX, aY: real);
var
 temp: real;
begin
 temp:= aX;
 aX:= aY;
```

```
  aY:= temp;
end;
procedure IntercambiaEnteros(var aX, aY: integer);
var
  temp: integer;
begin
  temp:= aX;
  aX:= aY;
  aY:= temp;
end;
procedure IntercambiaChar(var aX, aY: char);
var
  temp: char;
begin
  temp:= aX;
  aX:= aY;
  aY:= temp;
end;
procedure IntercambiaCadenas(var aX, aY: string);
var
  temp: string;
begin
  temp:= aX;
  aX:= aY;
  aY:= temp;
end;
end.
```

Arreglos lineales. Estructuras de alternativas y repeticiones

Introducción:

Hasta el momento se ha almacenado la información que se desea procesar en variables de algún tipo simple estándar. Pero, qué pasa si se necesita trabajar con una lista de elementos, por ejemplo una lista de edades de un grupo de personas, o de valores de temperatura máxima de cada día de un periodo, o de los pasajeros de un vuelo de avión, etc. Nótese que:

- la cantidad de elementos de las listas pudiera ser diferente. Por poner solamente un ejemplo, analícese que no todos los vuelos de avión tienen la misma cantidad de pasajeros.
- Si se necesita hacer varias operaciones con los elementos de la lista, por ejemplo, promediarlos y después contar los que están por encima del promedio. Será obligatorio tenerlos almacenados todos permanentemente, al menos mientras dure el procesamiento

Se comenzará el estudio de un nuevo tipo de datos que permite dar solución a este problema, el array o arreglo, también se abordan las estructuras de control de alternativas y repetitivas, o sea, aquellas que permiten escribir algoritmos no lineales. Como puede imaginarse, rara vez en la vida la solución de un problema implica una secuencia lineal de pasos. Por el contrario, lo más frecuente es encontrar algoritmos en los cuales es necesario hacer unos pasos u otros en dependencia de determinadas condiciones.

De ahí que, las distintas formas de representación del conocimiento usadas por el hombre incluyan la posibilidad de expresar condiciones lógicas, o sea proposiciones condicionales. Por citar sólo algunos ejemplos:

a) En lenguaje natural
   "*si llueve entonces me mojo*"
b) En lenguaje de la lógica proposicional
   p = *llueve*
   q = *me mojo*
   $p \rightarrow q$

Véase otro ejemplo retomando la frase estudiada en los temas de lógica:

"Existe algún hombre que es buen pelotero"

que utilizando el lenguaje de la lógica de predicados se puede expresar por:

P: predicado buen pelotero

$\exists x\ P(x)$

Si, dado un grupo de hombres se desea probar la validez de la expresión anterior, sería necesario ir comprobando para cada uno de ellos si cumple la condición ser buen pelotero. Nótese que este algoritmo implica repetir la comprobación hasta encontrar el primero que la satisfaga o hasta que ya no quede ninguno por chequear. Ante un algoritmo como este las estructuras secuencial y alternativas no son suficientes, hacen falta también estructuras que permitan repetir uno o varios pasos en un algoritmo.

Analícese, a modo de ejemplo, el algoritmo para calcular el factorial de un número natural que implica obtener el producto de todos sus predecesores. Para ello es necesario repetir la operación producto varias veces, en dependencia del número en cuestión. Este algoritmo podría plantearse como sigue:

Factorial de un número

```
Entrar número (n)
Si n >= 0
  Si n=0 o n=1
    Factorial= 1
   Si no
    Factorial=1
    Para i=1 Hasta n
      Factorial= Factorial * i
    Fin
  Fin
Si no
  Mostrar no tiene Factorial
Fin
Fin
```

Nótese que en este algoritmo se plantean alternativas de solución. Por ejemplo, si el número es mayor o igual que cero se procede de una forma, mientras que si es menor se procede de otra y cuando ya se ha decidido calcular el factorial hay que repetir una operación producto varias veces según el número que sea.

1.- El tipo Array

El array es un tipo de dato que permite almacenar una colección de elementos del mismo tipo, bajo un mismo identificador, y que permite tratar la colección como un todo y también acceder a cada elemento por separado. Los elementos de un arreglo se diferencian entre sí por la posición que ocupan dentro del mismo, la cual se expresa a través de uno o más subíndices.

Sintaxis

type

identificador= **array**[valor inicial del subíndice.. valor final del subíndice[,valor inicial del subíndice.. valor final del subíndice, ...]] **of** tipo de los elementos;

Aunque los arreglos pueden tener más de una dimensión, en este curso solamente se trabajará con arreglos de una sola dimensión.

Al definir los valores inicial y final del subíndice se está especificando el número de elementos en el arreglo. El índice puede ser un rango de cualquier tipo ordinal (excepto Longint).

Ejemplo

```
type
  TArreglo = array[1..100] of real;
var
  lista: TArreglo;
```

Esquematizar cómo quedarían los arreglos

1	**2**	**3**	**4**	**.....**								**100**	**← índice**
1.5	2.7	6.0	1.9	..	...	...	...	...					**←elementos**

Note que la memoria requerida por una variable de tipo array se reserva de manera estática, por lo que la memoria que no se utilice será desperdiciada y por eso en las últimas versiones aparece la posibilidad de asignar la memoria de manera dinámica en función de la cantidad real de elementos (arreglos dinámicos). En este curso no se trabajarán los arreglos dinámicos.

Acceso a los elementos de un array
Para acceder a los distintos elementos de un array se escribe el identificador de la variable y a continuación entre corchetes el valor del índice. Por ejemplo:

1. **if** lista[1]<=20.6 **then** lista[1]:=sqr(lista[1]);
2. **for** cuenta:=1 **to** 100 **do**
   if lista[cuenta] < 0 **then** Mostrar(cuenta, lista[cuenta]);

Los elementos individuales del array pueden ser utilizados en expresiones, sentencias de asignación, en E/S, etc., como si fueran variables simples.

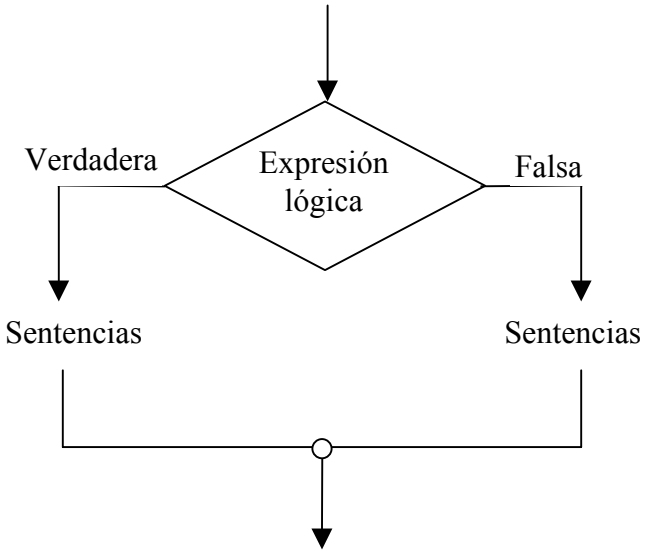
Pero ¡**CUIDADO**!
1) Una variable del tipo array no puede leerse ni visualizarse directamente, sino elemento a elemento.
2) Las variables del tipo array sólo pueden participar en una expresión aritmética elemento a elemento.
3) Se puede asignar una variable del tipo array a otra variable del tipo array, siempre que sean del mismo tipo.

Ejemplo
```
var
   A, B, C: TArreglo;
   D: array[1..100] of char;

1- edt1.text:= A;        NO!
   edt1.text:=A[i];      SI! Dentro de un lazo donde varía  i
2- C:= A + B;            NO!
   C[i]:=A[i] + B[i]; SI! Dentro de un lazo donde va variando i
3- A:= B;                SI! Son del mismo tipo, es lo mismo si se hace elemento a  elemento
   D:=C;                 NO! No son del mismo tipo
```

2.- Alternativa simple: sentencia If-Then-Else. Sentencia compuesta.

Una estructura alternativa es aquella que permite las sentencias que se deben ejecutar según el resultado de una expresión lógica. Por tanto, una estructura como esta, permite plantear algoritmos en los cuales existen alternativas de solución.

Está formada por una expresión lógica, seguida de una o varias sentencias subordinadas que se realizan si la expresión lógica es verdadera y opcionalmente otras que se realizan si la expresión lógica es falsa.
De una manera gráfica podría representarse como sigue:



Se puede formalizar a través del siguiente seudo código:

SI expresión lógica
Conjunto de sentencias a ejecutar si la expresión lógica es verdadera
SI NO
Conjunto de sentencias a ejecutar si la expresión lógica es falsa
FIN

Ejemplo:

```
SI Valor < 1000
  Suma = valor +3
   incremento = 0
SI NO
  Suma = valor * 0.01
```
FIN

Nótese que el algoritmo anterior implica:

- realizar más de una acción si la expresión lógica es verdadera.
- realizar uno de los dos conjuntos de operaciones solamente.

En Pascal existe la instrucción If-Then-Else para programar un segmento como el anterior. Con la siguiente sintaxis:

if exp. lógica **then** sentencia 1
else sentencia 2

donde tanto sentencia 1 como sentencia 2 pueden ser:

- simple: una sentencia formada por una sola instrucción.
- compuesta: una sentencia formada por más de una instrucción, separadas por punto y coma, ";", que se encierran entre las palabras reservadas **begin** y **end**.

El ejemplo anterior en Pascal se expresa como sigue:

```
if valor < 1000  then
 begin
   suma:= valor +3;
   incremento:= 0;
 end
 else
   suma:= valor * 0.01;
```

Antes de un **else** NUNCA se escribe punto y coma, ";" pues realmente el fin de la sentencia **if** es después de la sentencia que aparece a continuación del **else**. Pero entre las instrucciones de una sentencia compuesta sí.
Después de ejecutar un **else** el compilador ejecutará la sentencia siguiente al **if**. En el caso del ejemplo anterior se ejecutará

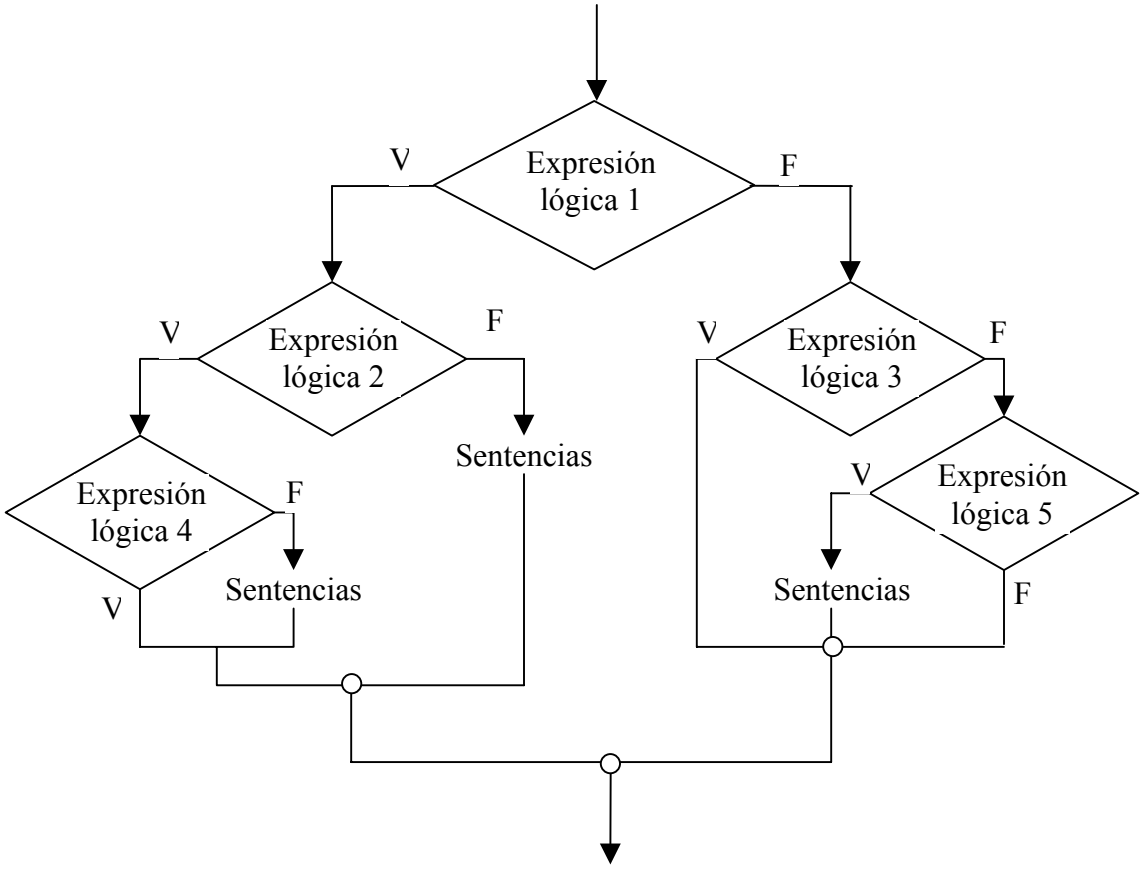
Otros ejemplos de alternativas simples:

```
if a < b then
  z:= k + 5; //si a es menor que b se ejecuta esta sentencia y después la que le siga

if  (c >= 12) and (c < 24) then
  h:= 2  //si la exp. es cierta se ejecuta esta sentencia, después la que sigue a h:=3;
else
  h:= 3; //si la exp. es falsa se ejecuta esta sentencia, después la que sigue a continuación

if  h < 0 then
 MostrarMensaje('La altura no puede ser negativa.'); //si h es menor que 0 se invoca al
                        //procedimiento MostrarMensaje al retornar se ejecuta la sentencia
                        //que le sigue a la invocación

if  h <= 0 then //en este ejemplo las dos sentencias son compuestas
  begin
     z:= k / 2;
     s:= s + 10;
  end
  else
```

```
begin
  z:= - k / 2;
  p:= s – 12;
end;
```

3.- Alternativa múltiple: sentencias If-then-else anidadas y sentencia Case.

Con mucha frecuencia la selección de una alternativa no es tan simple. A veces se requiere chequear más de una condición antes de realizar una operación concreta. A este tipo de situación se le reconoce como alternativa múltiple y se puede programar a través de if-then-else anidados o de la sentencia Case.
El ejemplo inicial muestra una alternativa múltiple. Si n>= 0 entonces es necesario chequear si n=0 o n=1 para asignar 1 al Factorial por definición o calcular el factorial por el producto de sus predecesores.
Gráficamente una situación como esta podría representarse de la siguiente manera:



Como el gráfico lo indica la sentencia a ejecutar por uno de los caminos de una alternativa, es otra alternativa y se puede programar anidando estructuras alternativas. Esto se puede expresar en un algoritmo de diferentes maneras.

Ejemplos

```
1.- if a < b then //verificar que esto no hace lo mismo que el ejemplo 2
     if c > d then // aquí si a >= b no se ejecuta nada
      z:= 5
      else
        z:= 7;
2.- if (a < b) and (c > d) then // aquí si a >= b se ejecuta z:= 7
      z:= 5
      else
        z:= 7;
3.- if n >= 0 then //verificar que esto es equivalente al ejemplo 4 que es más claro
      if (n=0) or (n=1) then
       Factorial:= 1 //no se pone ; pues está  delante de un else
      else
        begin
          Factorial:=1
          Factorial:=RepetirProducto(Factorial, n)
        end
    else
      Mostrarmensaje('No tiene Factorial');
 4.- if n < 0 then Mostrarmensaje('No tiene Factorial')
      else
        if (n=0) or (n=1) then
          Factorial:= 1
        else
          begin
            Factorial:=1
            Factorial:=RepetirProducto(Factorial, n)
          end
```

Sentencia Case

En ocasiones tenemos que decidir qué camino tomar en la solución de un problema teniendo en cuenta que se está a frente a un caso de muchos posibles. Como se vio anteriormente esto se puede programar con varios if- then – else anidados. Pero cuando existen varios niveles de anidamiento se compromete la claridad. Para esta alternativa múltiple Pascal cuenta con la estructura Case. Su sintaxis es la siguiente:

```
case selector of
  lista de constantes 1: [sentencia 1];
  lista de constantes 2: [sentencia 2];
   ...
  lista de constantes n: [sentencia n];
   [else
       sentencia k]
end;
```

donde:
lista de constantes i: una lista de posibles valores del selector.

En esta estructura de control el *selector* puede ser una variable, una constante o una expresión, en todos los casos de tipo **ordinal**. Con esta alternativa múltiple se va a verificar si el valor del selector coincide con un valor de alguna de las listas de valores posibles. Si el valor del selector se encuentra entre los especificados en la lista 1 se ejecutará la sentencia 1 y la ejecución continúa con la sentencia que sigue a la alternativa, o sea después del **end**. Si no, no se ejecuta la sentencia 1 y se pasa a verificar si coincide con algún valor de la lista 2, en dicho caso se ejecutará la sentencia 2 y la ejecución continuará con la instrucción que sigue a la alternativa, y así sucesivamente. Si no coincidiera con ninguna de las variantes y aparece la cláusula **else** de la estructura se ejecutarán una tras otra las instrucciones de la sentencia k. Si no coincide con ninguna de las variantes y no aparece cláusula **else**, entonces no se ejecuta ninguna de las sentencias que aparecen en la alternativa y se pasa a ejecutar la instrucción que sigue a la alternativa. Nótese que la alternativa múltiple termina con **end**. Por otro lado, cualquiera de las sentencias puede ser simple o compuesta.

En las listas de constantes no pueden aparecer variables. Ellas son constantes de tipo ordinal y constituyen posibles valores del selector. Dentro de una lista, las constantes se separan por coma. Una misma constante no puede aparecer en más de una lista.

Ejemplos de alternativa múltiple

```
case A + B of
    -1, 0, 1:  MostrarMensaje('El valor es muy bajo.');
        2, 3: MostrarMensaje ('El valor es bajo.');
           4: begin
                MostrarMensaje ('El valor es moderado. Debe ingresar el próximo dato: ');
                CaptarDato(ProxDato);
              end //recuérdese que delante de un else nunca se escribe ;
 else
   MostrarMensaje ('El valor es alto o muy alto y será disminuido.');
   a:= a – 2;
end;
```

En el ejemplo se ha asumido que las variables *a* y *b* son de tipo entero, luego su suma es de tipo entero, o sea, ordinal. Aquí se informa al usuario que el valor es muy bajo si el valor de *a* + *b* está entre –1 y 1. Si el valor es 2 ó 3 se informa al usuario que el mismo es bajo. Si fuera 4 se anuncia que el valor es moderado y se solicita del usuario el próximo dato. Si el valor no está en ninguno de los anteriores casos se informa que es muy alto y se disminuye el valor de la variable *a*.

Es necesario llamar la atención sobre lo siguiente, cualquier alternativa que se puede programar con *case*, también se puede programar con *if – then- else* anidados pero lo contrario no se cumple, pues siempre las expresiones lógicas a verificar no son chequear si una expresión ordinal tiene un determinado valor o no.

4.- Estructura repetitiva con variable de control (For)

En el ejemplo del cálculo del factorial todavía queda por definir cómo se puede expresar a través de una estructura de control el producto acumulativo. O sea, ahora se habla de una estructura que permita realizar varias veces una o varias sentencias.
A estas estructuras se les conoce como estructuras repetitivas o iterativas.
Este tipo de estructura define un ciclo en el algoritmo y como todo ciclo debe tener una forma de parar, para permitir continuar el algoritmo, de lo contrario la secuencia se convertiría en infinita y por tanto dejaría de ser un algoritmo (recuérdese que una propiedad de todo algoritmo es que debe ser finito)
Nótese que en el caso especifico del factorial se necesita hacer una operación para todos los valores de un rango dado, o sea para todo i desde 1 hasta n. En otras palabras en un caso como este se conoce de ante mano la cantidad de veces que se requiere realizar las operaciones a repetir y por tanto se sabe cuando debe para el ciclo.

Utilizando seudo código se puede formalizar de la siguiente manera:
PARA valor = valor inicial **HASTA** valor final
sentencia 1
    sentencia 2
    ....
    sentencia n
FIN

Ejemplo
PARA i= 1 **HASTA** n
    Factorial:= Factorial * i
FIN

En Pascal una estructura como esta se escribe con la sentencia for – do que tiene la siguiente sintaxis:
a) **for** variable de control:= valor inicial **to** valor final **do** sentencia
b) **for** variable de control:= valor inicial **downto** valor final **do** sentencia

La variable de control debe ser un identificador de variable de cualquier tipo ordinal. El valor inicial y final deben ser de un tipo compatible con el tipo ordinal y pueden ser expresiones. A la variable de control siempre se le asigna al comienzo el valor inicial. Para el caso a) el valor de la variable de control se incrementa en 1 para cada repetición. Si el valor inicial es mayor que el final no se ejecuta la sentencia. Para el caso b) el valor de la variable de control es decrementado en 1 para cada repetición. Si el valor inicial es menor que el valor final la sentencia no se ejecuta.

Causa un error si en la secuencia de instrucciones se altera el valor de la variable de control. La variable de control toma automáticamente el valor siguiente cada vez que se repite la secuencia de instrucciones hasta que exceda en 1 al valor final. La variable de control al terminar el ciclo **for-do** tiene un valor mayor que el valor final si es con **to** y menor si es con **downto**. En cualquiera de los dos casos si el valor inicial es igual al valor final se ejecuta una vez la sentencia. La sentencia puede ser simple o compuesta.

Ejemplos

```
a) for i:= 2 to 63 do
      if  x>max then max :=x ; //una sentencia simple que se repite 62 veces
c) var i: integer;
    .....
    for i:= 0 downto -12 do //una sentencia compuesta que se repite 13 veces
      begin
        z:= 2 * i + 5;
        Mostrar(i , z);
      end;
    for i:= 1 to n do
       Factorial:= Factorial * i;// en Factorial va quedando el producto acumulado
k:=0;
   suma:= 0;
   for i:=1 to 100 do
 begin //sentencia compuesta
   CaptarDato(x);
       if x > 20 then
         begin
           k:= k+1;// se van contando los que son mayores que 20
          suma:= suma +x;//se van sumando los que son mayores que 20
         end;
     end;
   PromedioMayoresQue20:= suma/k;//calcula el promedio de los que son mayores que 20
```

5.- Estructuras repetitivas con condición

No siempre se conoce la cantidad de veces exactas que se va a repetir un conjunto de pasos. Por el contrario, existen situaciones donde la repetición es controlada por condiciones explícitas que deben cumplirse al principio o al final, para estos casos se utilizan los formatos de REPETIR y MIENTRAS.

REPETIR	MIENTRAS **exp. lógica**
pasos	pasos
HASTA exp. lógica	**FIN**

En el formato de la izquierda la expresión lógica se chequea al final de la iteración, lo cual garantiza que la secuencia de pasos se realiza al menos una vez (poschequeo de condición). En el formato de la derecha la condición se chequea al principio, por lo que puede que la secuencia de pasos no llegue a realizarse ni siquiera una vez (prechequeo de condición).
Un algoritmo puede ser descrito utilizando más de una construcción (permitiéndose los anidamientos) siempre y cuando tengan una única entrada y una única salida, para garantizar la estructuración lógica del mismo.
A continuación se verá la sintaxis de cada una en Pascal.

- **While**

Las estructuras repetitivas con precondición se representan en Pascal por la palabra reservada **While** y responden a la siguiente sintaxis:

while expresión lógica **do**
    sentencia;

Con esta estructura se verifica, primero, si la expresión lógica se cumple, en dicho caso se ejecutará la sentencia (que puede ser simple o compuesta) y se vuelve a verificar la expresión lógica. Mientras la expresión lógica sea verdadera se estará ejecutando la sentencia. Resulta obvio que entre las instrucciones de la sentencia debe haber una que implique un cambio del valor de la expresión lógica. De lo contrario, nunca se dejaría de ejecutar la secuencia o, lo que es lo mismo, se caería en un ciclo infinito.

Por la forma en que se ejecuta esta estructura de control, pudiera darse el caso de que al llegar al ciclo la expresión lógica sea falsa de antemano. Esto conllevará a que la sentencia nunca se ejecute.

Ejemplo del uso de estructura repetitiva con precondición:

```
a) i:= 1; factorial:=1 //fragmento equivalente al programado con for
   while 1 <= n do
     begin //una sentencia compuesta
       factorial:= factorial * i;
     inc(i); //aquí el programador es responsable de incrementar la variable de control, a
             //diferencia del for, de lo contrario se hace infinito el ciclo
   end;
b)y:= ValorInicial;
  while y > Error do //nunca se realiza si ValorInicial <= Error
    y:= y/Delta; //y cada vez es menor, por lo que se garantiza la condición de parada

c)cadena:= 'Pascal';
  salir:= false;
  i:=1;
  while (i<=ultimo) and   not salir do //busca si existe al menos un nombre igual a Pascal
     begin
       salir:=  DevuelveNombre(i) = cadena;// se asigna a la variable booleana salir el valor
       inc(i);                                  //de la expresión lógica
     end;
```

En este último ejemplo la sentencia de asignación que se hace dentro del ciclo es equivalente a:

```
if DevuelveNombre(i) = cadena then
  salir:= true
 else
   salir:= false;
```

Nótese que es imprescindible que al inicio de la estructura estén creadas las condiciones para poder evaluar la expresión lógica, o sea todas las variables que intervienen en ella deben haber tomado algún valor.

- **Repeat**

Las *estructuras repetitivas con postcondición* se programan en Pascal mediante la sentencia **repeat - until** y responde a la siguiente sintaxis:

```
repeat
    sentencia
until expresión lógica = verdadera;
```

Con esta instrucción se ejecuta la sentencia y luego se verifica si expresión lógica es verdadera. Si la expresión lógica es falsa se pasa a repetir de nuevo la sentencia y así sucesivamente. Esto significa que hasta que no se cumpla la condición de parada se estará repitiendo la sentencia. En esta estructura la sentencia siempre se ejecuta al menos una vez. Será exactamente una vez si luego de la ejecución de la secuencia la condición de parada se cumple.

En la sentencia debe haber una instrucción en la que cambie el valor de la expresión lógica, para garantizar que en algún momento se alcance la condición de parada. La sentencia nunca se encierra entre **begin** y **end**.

Ejemplo del uso de estructura repetitiva por poscondición:

```
a)  Factorial:=1;
    i:=1;
    repeat
    factorial:= factorial * i;
    inc(i);
    until i>n;
b)repeat
    CaptarDato(n); se invoca
  until (n>=0) or (n<=9); //n toma valor por primera vez dentro del ciclo, a través de la
                          //invocación a CaptarDato
c)Cadena:= 'Pascal';
  i:=1;
  repeat do //busca si existe al menos un nombre igual a Pascal
    salir:=  DevuelveNombre(i) = Cadena;// se asigna a la variable booleana salir el valor
    inc(i);                             //de la expresión lógica
  until (i>ultimo) or salir;
```

Nótese que este fragmento hace lo mismo que el último ejemplo del **while**

6.- Ejemplo.
Para concluir se programará un subprograma que calcule el factorial de un número.

```
function Factorial(n: integer):double; //porque puede que el  resultado no quepa
var                                    // en un entero
  i: integer;
begin
if n<0 then
  result:= 0// si la function devuelve cero es que no se pudo calcular el factorial
else
  begin
    result:= 1;
    if (n<>0) and (n<>1) then
      for i:= 1 to n do
        result:= result  *  i;
  end;
end;
```

Este es el algoritmo planteado inicialmente, pero puede programarse la solución de este problema de una manera más eficiente. Véase a continuación:

```
function Factorial(n: integer):double; //porque puede que el  resultado no quepa
var                                    // en un entero
  i: integer;
begin
if n<0 then
  result:= 0// si la function devuelve cero es que no se pudo calcular el factorial
else
  begin
    result:= 1;
    for i:= 2 to n do //si n es 0 o 1 no entra al ciclo y se devuelve 1
      result:= result  *  i;
  end;
end;
```

Dividir una lista en dos, por la mitad, si es posible informando sobre el éxito de la operación

Tipos de datos simples definidos por el programador. Algoritmos básicos
Introducción:

En este teme dos tipos de datos, el subrango y el enumerado, que el programador puede declarar en sus programas Pascal con el objetivo de aumentar la claridad de éstos. Estos tipos de datos no se consideran indispensables para la programación por lo que no se encuentran frecuentemente implementados en los lenguajes, no obstante poder trabajar con ellos es de una gran ventaja, pues permiten acercar más los datos con los que se trabaja en el programa a los datos reales con los que se trabaja en la vida real.

Para que se tenga un ejemplo, supóngase que se está programando una aplicación para la planificación de horarios docentes. En este caso será necesario manipular los días de la semana. Para ello se puede proceder de dos formas, la primera codificar los días por ejemplo:
Lunes—1
Martes—2
Miércoles —3
...
Domingo—7
de manera que a través de una variable numérica cuyo contenido sea el código correspondiente a un día se pueda saber de cual día se trata, por ejemplo:

```
function DiaSemana(aDia: byte): string;
begin
  case dia of
      1: result:= 'Lunes';
      2: result:= 'Martes';
      3: result:= 'Miércoles';
      4: result:= 'Jueves';
      5: result:= 'Viernes';
      6: result:= 'Sábado;
      7: result:= 'Domingo;
  end;
end;
```

Nótese que en este caso aunque se utiliza una variable numérica para almacenar un código, la misma no debe participar en ninguna operación numérica, lo cual sería responsabilidad del programador garantizar. Si se tiene en cuenta que la modularidad permite que varios programadores trabajen en una misma aplicación, el peligro de que esto ocurra es evidente.
La segunda forma es, tratar los días por sus propios nombres, siempre que se necesite trabajar con ellos. Si se comparan ambas formas es evidente que la segunda es mucho más clara.
Otra situación frecuente es tratar con un tipo de datos cuyo rango de valores posibles no se ajusta exactamente a ninguno de los tipos estándar definidos en el lenguaje.

Tipo de dato subrango

En algunos problemas, se trabaja con datos que se mantienen en un rango de posibles valores que no coincide con ninguno de los estándar. Ejemplos clásicos son los días del mes que varían en el rango de 1 a 31 y la edad de una persona que varia entre 0 y 150 años aproximadamente.

Para estos casos el Pascal cuenta con el *Tipo Subrango*, llamado así porque los valores involucrados pertenecen a un subrango de algún tipo ordinal.
Los valores definidos en el tipo subrango tienen un tipo base, que tiene que ser ordinal y en el momento del tratamiento se cuenta con todas las facilidades con que cuenta el tipo base, o sea, si los valores del subrango son enteros, se pueden aplicar operaciones aritméticas sobre ellos, si los valores son de carácter se pueden aplicar sobre ellos todas las operaciones de caracteres, etc.

Sintaxis

```
type
  identificador= valor inicial del subrango..valor final del subrango;
```

También se pueden declarar variables de algún tipo subrango directamente.

```
var
  identificador: valor inicial del subrango..valor final del subrango;
```

En este último caso, por supuesto, no se podrá utilizar el identificador declarado para definir nuevos tipos o variables.

Ejemplos

```
const
max= 100;
type
  TDiaMes= 1..31;//Tipo base: números  enteros
  TMayuscula='A'..'Z'; //Tipo base: conjunto de caracteres
1.1.1.1.1.1.1.1   TMes= 1..12; //Tipo base: números  enteros
  TRango= 1..max;
var
  Ayer, Hoy, UltimoDia: TDiaMes;
  Caracter: TMayuscula;
  Lista: array[TRango] of integer; {los valores de los índices de los arreglos se definen por subrango, ya sean directamente o a través de un identificador de tipo subrango declarado previamente}
```

Nota: en lo adelante se usará como convenio escribir una T mayúscula delante de los identificadores de tipo.

El siguiente ejemplo muestra el uso de algunos de los tipos subrango definidos arriba, en una función.

```
function   DevuelveUltimoDia(aMesActual: TMes; aAnoActual: integer): TDiaMes;
//Devuelve el último día de un mes conociendo el número del mes y el año
begin
  case aMesActual of
    1, 3,  5,  6, 8, 10, 12: result:=31;
    2: begin
       if (1980-aAnoActual) mod 4 = 0 then
            result:= 29    //1980 fue un año bisiesto. Si el resto de la división entre 4 de la
          else             //diferencia entre 1980 y el año actual es 0 se puede asegurar que el
          result:= 28;     //año actual es también bisiesto
        end
   else
     result:= 30;
  end;
end;
```

Tipo de dato enumerado o numerativo

El tipo enumerado permite dar solución al problema de los días de la semana planteado en la introducción. Cuando se quiere manipular este tipo de información en un programa, representarla en forma de números enteros, o caracteres tiene las siguientes desventajas:

- la forma no es natural, o sea es bastante alejada de como lo hace el hombre.
- no hay garantías de que una variable que los almacene no sea utilizada en una operación aritmética o de cadena según sea el caso.

Para evitar estas dificultades el Pascal brinda al programador la posibilidad de definir sus propios datos a través del *Tipo Enumerado*. De esa forma, podremos definir, como dato para una variable, el color de las luces del semáforo, los días de la semana, los meses del año, etc.

Sintaxis

```
type
  Identificador = (valor1, valor2 [, valorn …]);
```

Ejemplo

Considérese la siguiente declaración de tipos y variables:

```
type
   TNombreMes = (Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov, Dic);
   TDiaSemana = (Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo);
   TLuzSemaforo = (Roja, Amarilla, Verde);
   TOperación = (Suma, Resta, Multiplicacion, Division);

var
   MesActual, UltimoMes: TNombreMes;
   DiaActual: TDiaSemana;
   LuzEncendida, LuzApagada: LuzSemaforo;
   Operador: Operación;
   HorasExtras: array[lunes..domingo] of real;
```

El tipo enumerado es:

- simple: de ahí que las variables de ese tipo toman un único valor en cada momento.
- ordinal: por lo que en el ejemplo de los meses, el valor Feb es el sucesor del valor Ene y el valor Verde es mayor que el valor Rojo.

Es necesario aclarar que todos los valores definidos en el tipo enumerado, por ejemplo, Ene, Feb, Mar, Lunes, Martes, Rojo, etc., deben ser distintos entre sí y para el compilador se comportan como identificadores. Debido a esto pueden escribirse lo mismo con mayúsculas que con minúsculas como en el caso de la variable HorasExtras, donde para definir el subrango del índice se han escrito con minúsculas.

La variable MesActual declarada en el ejemplo tomará en cada instante de ejecución, exactamente uno solo de los valores Ene a Dic y ningún otro tipo de valor, como mismo una variable real no puede tomar como valor un dato de tipo *Char*.

Operaciones definidas sobre el tipo enumerado en Pascal.

- Variables de tipo enumerado no pueden ser leídas ni escritas con *Read* ni *Write*, respectivamente, pues el contenido es precisamente un identificador.
- Variables de tipo enumerado sólo pueden ser asignadas a variables del mismo tipo enumerado. Nunca a cadenas de caracteres, luego no se pueden asignar a una propiedad tipo Text de algún componente o como parámetro de un ShowMessage, etc.
- Los valores definidos para un enumerado pueden ser utilizados dondequiera que pueda utilizarse un ordinal.

Ejemplos

- UltimoMes := MesActual;
- DiaDeHoy := Miercoles;//no se ponen apóstrofos pues los valores son enumeradas.
- LuzApagada := Succ(LuzEncendida);
- MesActual := Pred(UltimoMes);
- **for** DiaDeHoy := Lunes **to** Viernes **do** …;
- **case Operador of**
  Suma: resultado:= x+y;
  Resta: resultado:= x-y;
  end;
- **type**
  color = (rojo,blanco,azul,amarillo,verde);
  var
  indice : color;
  valores : **array**[color] **of integer**;
  colores : **array**[1..100] **of** color;
  .....
  for indice:=rojo **to** verde **do**
  Mostrar (valores[indice]);

- Se pueden comparar entre sí.

Ejemplos

- **If** DiaDeHoy = Miercoles **then** ...
- **If** MesActual < UltimoMes **then** ...

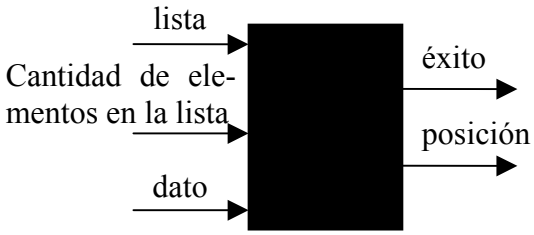
- La función *Ord* con un identificador de enumerado como argumento, devuelve el ordinal equivalente al mismo. El compilador asocia ordinal 0 al primer elemento definido en el enumerado, 1 al siguiente, y así sucesivamente.

1. Algoritmos básicos.

A continuación se estudiarán algunos algoritmos que todo programador debe conocer. Algunos de ellos se vieron en el tema de algoritmización. Se partirá en cada caso de un problema concreto. En la semana anterior se comenzó la programación de una unit para la manipulación de una lista de números enteros, que ahora se retomará para incluir otras operaciones necesarias.

- Buscar un valor dato y si se encuentra devolver su posición.
- Insertar un elemento en una posición intermedia de la lista.
- Eliminar un elemento informando sobre el éxito de la operación.
- Convertir cada uno de sus elementos a cadena y devolver una nueva lista con ello.
- Crear una nueva lista con los elementos pares.
- Crear una nueva lista con los elementos primos.
- Buscar el menor elemento de la lista
- Buscar un valor dato y si se encuentra devolver su posición.

El módulo, visto como una caja negra, debe recibir una lista y su cantidad de elementos, así como, un valor a buscar. A partir de esa información, debe responder si se encontró el valor buscado y en caso positivo devolver la posición donde está.



Es necesario destacar que:
Este módulo tiene dos resultados, pero uno de ellos, el éxito de la búsqueda, se entrega siempre mientras que el otro, la posición, se da sólo si se encuentra el dato. Por este motivo el módulo debe ser una función booleana que tenga además un parámetro a través del cual se informe la posición en caso de encontrarse.
La búsqueda se hace comparando cada elemento de la lista con el dato, desde el primero hasta que se encuentre o hasta que ya no queden elementos con los cuales comparar, o sea hasta que se alcance el final de la lista. Por supuesto lo mismo sería comenzar desde le final y avanzar hacia el principio de la lista, pero esta no es la forma natural de realizar una búsqueda en una lista cualquiera. A este método de búsqueda se le conoce como búsqueda secuencial.

Precondiciones:
Se tiene una lista de elementos
La cantidad de elementos de la lista es un valor mayor o igual que cero
El dato a buscar es un valor del mismo tipo de elementos de la lista

Algoritmo en pseudocódigo:
Búsqueda de un elemento en una lista

ENTRAR Dato, Lista, Cantidad
Posicion= 1
Encontrado= Falso
MIENTRAS Posicion <= Cantidad **Y** No Encontrado
  SI Elemento $_{Posición}$ = Dato
    Encontrado= Verdadero
  SI NO
    Posicion:= Posicion + 1
  FIN
 FIN
 MOSTRAR Encontrado, Posición
FIN

Poscondiciones
Si la búsqueda fue exitosa encontrado es verdadera y se conoce la posición del elemento en la lista
Si la búsqueda fracasa no tiene sentido hablar de la posición
La lista queda como al inicio, o sea no sufre ningún cambio

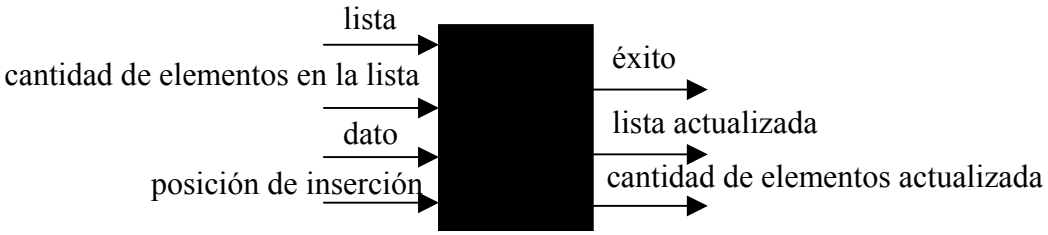
En Pascal

```
Function BuscaElemento(aLista: TArregloEntero; aCantidad: TRango; aDato: integer; var
Posicion: TRango): boolean;
Begin
  result:= false; //al inicio no se ha encontrado el elemento
  Posicion:= 1;
  while (Posicion <= aCantidad) and not result do //contiene una sentencia simple
     if aLista[Posicion] = aDato then
       result:= true //sólo si se encuentra se cambia el valor de result y no se entra más al ciclo
     else
       Posicion:= Posicion + 1; //equivalente a inc(Posicion);
end;
```

Nótese que no es necesario chequear si la lista está vacía, pues en este caso no entraría al ciclo y por tanto se informa búsqueda fracasada.
Como en muchos casos existen otras formas de programar esto mismo, pero esta es la más clara y elemental.

Insertar un elemento en una posición intermedia de la lista.

El módulo, visto como una caja negra, debe recibir una lista y su cantidad de elementos, así como, un valor a insertar. A partir de esa información, debe responder si la operación fue exitosa y la lista actualizada.



Es necesario destacar que:
Este módulo al igual que el anterior debe ser una función booleana que tenga la lista como un parámetro de entrada / salida.
Es necesario chequear si la lista está llena, en cuyo caso la operación de inserción fracasa.
Es necesario chequear si la posición de inserción es válida. Se considera válida si se pretende insertar en una posición intermedia (posición<=cantidad) o al final de la lista (posición= cantidad + 1).
Para poder insertar será necesario correr todos los elementos desde la posición de inserción hasta el último, una posición a la derecha, con vistas a dejar el espacio de la posición de inserción vacío.

Precondiciones:
Se tiene una lista de elementos
La cantidad de elementos de la lista es mayor o igual que cero
Se conoce que el dato a insertar es del mismo tipo de los elementos de la lista
Se conoce que la posición de inserción es un valor mayor o igual que 1
Se conoce la máxima cantidad posible de elementos

Algoritmo en pseudocódigo:
Insertar un elemento en una posicion dada de una lista
ENTRAR Cantidad, Cantidad máxima, Lista, Posición, Dato
SI Cantidad < Cantidad máxima **Y** Posición <= Cantidad + 1
  PARA i = Cantidad **HASTA** Posición
    Lista $_{posición+1}$= Lista $_{posición}$
  FIN
  Lista $_{posición}$= dato
  Cantidad = Cantidad + 1
  MOSTRAR operación exitosa y lista actualizada
SI NO
  MOSTRAR operación fracasada
FIN

Poscondiciones
Se conoce si la operación de inserción fue exitosa o no
Si la operación de inserción fue exitosa lista queda actualizada y su cantidad de elementos incrementada
Si la operación de inserción fracasa no tiene sentido hablar de la lista actualizada

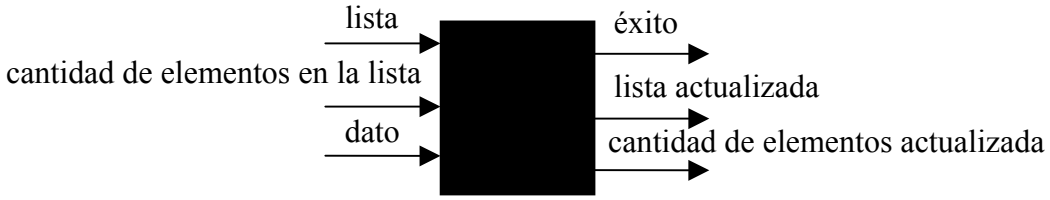
En Pascal

```
function InsertaElemento(var Lista: TArregloEntero; var Cantidad: TRango; aDato: integer; aPosicion: TRango): boolean;
var
  i: TRango;
Begin
  result:= false; // con esto se evita la cláusula else del siguiente if
  if (not ListaLLena) and (aPosicion <= Cantidad +1) then
    begin
      result:= true; // mejor al inicio para que no se olvide
      for i:= aCantidad downto aPosicion do //se abre un espacio en aPosicion
        Lista[i+1]:= Lista[i]; //sentencia simple
      Lista[aPosicion]:= aDato; //se coloca aDato en el espacio abierto o después del último
      Cantidad:= Cantidad + 1; // esto no puede hacerse antes del ciclo
  end;
end;
```

Nótese que en el caso extremo que se desee insertar antes del último, el algoritmo funciona, así como en el caso de que se desee insertar al final.

Eliminar un elemento informando sobre el éxito de la operación.

El módulo, visto como una caja negra, debe recibir una lista y su cantidad de elementos, así como, un valor a eliminar. A partir de esa información, debe responder el éxito de la operación y los datos de entrada actualizados.



Es necesario destacar que:
Este módulo al igual que los anteriores debe ser una función booleana que tenga la lista como un parámetro de entrada / salida.
Es necesario chequear si la lista está vacía, en cuyo caso la operación de eliminación fracasa.
Una vez encontrado el elemento a eliminar solamente se es necesario correr una posición hacia la izquierda los restantes elementos de la lista. Si el elemento es el último, solamente se necesita actualizar la cantidad de elementos de la lista.

Precondiciones:
Se tiene una lista de elementos
La cantidad de elementos de la lista es mayor o igual que cero
El dato a eliminar es del mismo tipo de los elementos de la lista

Algoritmo en pseudocódigo:
Eliminar un elemento
ENTRAR Lista, Cantidad, Dato
Buscar el elemento y obtener su posición
SI encontrado
  PARA i = posición **HASTA** Cantidad
$Lista_i = Lista_{i+1}$
 FIN
Cantidad = Cantidad -1
MOSTRAR encontrado
FIN

Poscondiciones
Se conoce si la operación fue exitosa o no
Si la operación fue exitosa se obtiene una la lista actualizada y la nueva cantidad de elementos de la misma.

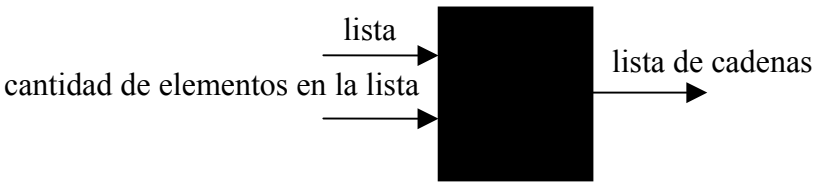
En Pascal

```
Function EliminaElemento(var Lista: TArregloEntero; var Cantidad: TRango; aDato: integer): boolean;
var
  i, p: TRango;
begin
 result:= false; // evita la cláusula else del if siguiente
 if BuscaElemento(Lista, Cantidad, aDato, p) then
   begin
     result:= true; //al inicio para que no se olvide
     for i:= p to Cantidad  - 1 do //se resta 1, para no acceder a un elemento que no existe.
       Lista[i]:= Lista[i+1]; //sentencia simple
     Cantidad:= Cantidad – 1; //esto no se puede hacer antes del ciclo
   end;
end;
```

Nótese que no es necesario chequear si la lista está vacía, pues en este caso BuscaElemento devuelve false y se detiene el proceso. Por otra parte, es necesario que el valor final de la variable de control del ciclo sea Cantidad – 1, para que dentro del ciclo cuando se acceda al elemento Lista[i + 1], no se trate de alcanzar un elemento que no existe.

Convertir cada uno de sus elementos a cadena y devolver una nueva lista con ello.

El módulo, visto como una caja negra, debe recibir una lista y su cantidad de elementos, y devolver una nueva lista con los elementos como caracteres.



Es necesario destacar que:
Este módulo al igual que los anteriores debe ser una función que devuelva la nueva lista.
La cantidad de elementos de la nueva lista es igual a la de la lista de entrada, por lo que no es necesario devolver esa información.

Precondiciones:
Se tiene una lista de elementos en Lista
La cantidad de elementos de la lista es mayor o igual que cero

Algoritmo en pseudocódigo:
Convertir una lista numérica a una lista de caracteres
ENTRAR Lista, Cantidad
PARA i = 1 **HASTA** Cantidad
  Lista de Caracteres[i] = cadena correspondiente al elemento Lista[i]
FIN

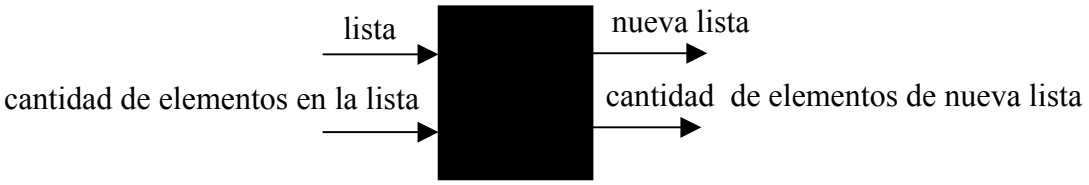
Poscondiciones
Se obtiene una lista de caracteres
La cantidad de elementos de la nueva lista es igual a la cantidad entrada
La lista inicial se mantiene igual que a la entrada

En Pascal
```
Function ConvierteAListaDeCadenas(Lista: TArregloEntero; Cantidad: TRango):
TArregloCadena;
var
   i: TRango;
begin
  for i:= 1 to Cantidad  do
     result[i]:= IntToStr(Lista[i]); //equivalente a Str(Lista[i],result[i])
end;
```

Nótese que:
no es necesario chequear si la lista está vacía, pues en este caso no se entra al ciclo.
La misma variable de control del lazo se utiliza para los dos arreglos.
A estos arreglos se les conoce como arreglos paralelos, pues en los dos los elementos de una misma posición guardan diferentes propiedades de una misma entidad. Los arreglos paralelos, por supuesto, siempre tienen la misma cantidad de elementos.

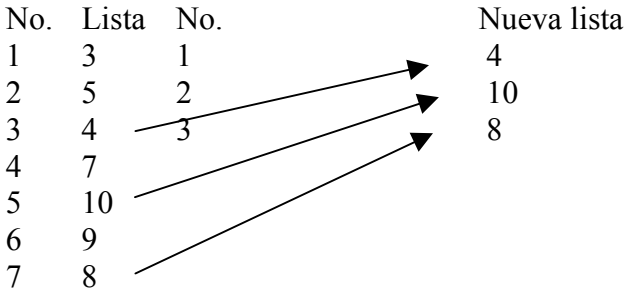
Crear una nueva lista con los elementos pares.

El módulo, visto como una caja negra, debe recibir una lista y su cantidad de elementos, y devolver una nueva lista con los elementos pares.



Es necesario destacar que:
Este módulo tiene, obligatoriamente, que devolver la nueva lista y su cantidad de elementos, que no tiene por qué coincidir con la cantidad de elementos de la lista de entrada.

Ejemplo



Precondiciones:
Se tiene una lista de elementos
Se conoce que la cantidad de elementos de la lista es mayor o igual que cero

Algoritmo en pseudocódigo:
Devolver una lista con los elementos pares de una lista original

```
ENTRAR Lista, Cantidad
Nueva cantidad = 0
PARA i = 1 HASTA CAntidad
  SI Lista[i] es par
    Nueva cantidad = Nueva cantidad + 1
    NuevaLista [Nueva cantidad] = Lista[i]
  FIN
FIN
MOSTRAR Lista
FIN
```

Poscondiciones
Se obtiene una nueva lista con los elementos pares de una lista de entrada
La cantidad de elementos de la nueva lista es mayor o igual a la cantidad de la lista inicial
La lista inicial se mantiene igual que a la entrada

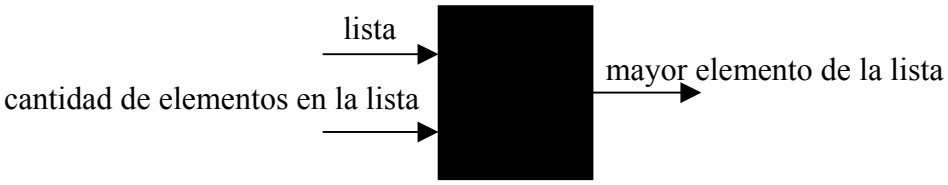
En Pascal

```
procedure CreaListaConPares(aLista: TArregloEntero; aCantidad: TRango; var NuevaLista: TArregloEntero; var NuevaCantidad: TRango);
var
  i: TRango;
begin
  NuevaCantidad:= 0;
  for i:= 1 to aCantidad  do
    if aLista[i] mod 2 = 0 then //sentencia simple
      begin
            NuevaCantidad:= NuevaCantidad +1;
       Nuevalista[NuevaCantidad]:= Lista[i];
      end;
 end;
```

Nótese que:
no es necesario chequear si la lista está vacía, pues en este caso no se entra al ciclo.
La lista de entrada se recorre completa
Para el subíndice de la lista de entrada se utiliza la variable de control del lazo, mientras que para la nueva lista se utiliza NuevaCantidad. Estos arreglos no son paralelos y no siempre tienen la misma cantidad de elementos. Solamente en el caso extremo en que todos los elementos de la lista inicial sean pares, los dos arreglos tendrán la misma cantidad de elementos.
NuevaCantidad se inicializa con 0 y solamente se incrementa cuando se encuentran elementos pares en la lista. El incremento debe realizarse antes de la asignación del elemento a la nueva lista, para garantizar que quede en el lugar adecuado. Si no se encuentran elementos pares NuevaLista se devuelve vacía y NuevaCantidad en 0, pues nunca se incrementó.

Buscar el menor elemento de la lista

El módulo, visto como una caja negra, debe recibir una lista y su cantidad de elementos, y devolver el mayor valor de la lista.



Es necesario destacar que:
Este módulo tiene que devolver un solo resultado, el mayor elemento de la lista.

Precondiciones:
Se tiene una lista de elementos en Lista
Se conoce la cantidad de elementos de la lista (Cantidad)

Algoritmo en pseudocódigo:
Este algoritmo toma un elemento de la lista y asume que es el menor. Después va comparando cada uno de los restantes elementos con el que hasta el momento de la comparación se considera el menor y cada vez que se encuentre uno que resulte más pequeño se actualiza el menor. Al finalizar el recorrido de la lista se tendrá el menor elemento de la misma.

```
Buscar el menor valor
ENTRAR Lista, Cantidad
Menor = Lista[1]
PARA i =2 HASTA Cantidad
  SI Lista[i] < Menor
    Menor = Lista[i]
  FIN
FIN
MOSTRAR Menor
FIN
```

Poscondiciones
Se conoce el menor valor de la lista de entrada

En Pascal

```
function Menor(aLista: TArregloEntero; aCantidad: TRango): integer;
var
  i: TRango;
begin
result:= aLista[1];
 for i:= 2 to aCantidad  do
    if aLista[i] < result then //sentencia simple
       result:= aLista[i]; //al encontrar uno más pequeño se actualiza menor
  end;
end;
```

Nótese que:
La lista de entrada se recorre completa
Al finalizar el recorrido de la lista en menor queda el valor más pequeño de todos
Con este algoritmo no se conoce la posición donde se encuentra el menor hallado. Puede suceder que el menor valor se encuentre en varias posiciones de la lista.

BIBLIOGRAFÍA

1. Matemática Discreta. (Capítulos 1. Lógica Matemática. Capítulo 3. Algoritmos).
2. Bartolomé José C. ‘’Lógica Informática’’, Tomos 1 y 2.
3. Bueno Eramis, García Luciano. ‘’Introducción a la lógica Matemática’’
4. Dr. Rosete Suárez Alejandro y otros. Clases de Asignatura “Lógica y Algoritmos”. 2002