

CCUZ 19

**LA PROGRAMACION**  
**EN**  
**LENGUAJE PASCAL**

**CENTRO DE CALCULO**  
**UNIVERSIDAD DE ZARAGOZA**

**ENERO - 1988**

## **LA PROGRAMACION EN LENGUAJE PASCAL**

**A. Salas Ayape**

Esta publicación es el texto correspondiente al curso LEG.01 que se imparte en el Centro de Cálculo de la Universidad de Zaragoza. Se estudian los elementos fundamentales del lenguaje, las estructuras de datos y las estructuras de control, ofreciendo una visión completa de sus posibilidades

CENTRO DE CALCULO DE LA UNIVERSIDAD DE  
ZARAGOZA, 1988

Ciudad Universitaria  
Edificio de Matemáticas  
50009 – ZARAGOZA  
Tfno.551278

Depósito Legal: Z-1093-88  
ISBN: 84-7733-056-5

El autor agradece la colaboración de Ma. Luisa Salazar en la edición del texto y expresa la voluntad de incorporar todas aquellas sugerencias que aporten los lectores para facilitar una comprensión más rápida y completa de la materia.

**CONTENIDO**

1.	INTRODUCCION	1 - 1
	Obtener información a partir de datos	1 - 2
	El arte de la buena programación	1 - 3
2.	ELEMENTOS BASICOS	2 - 1
	2.1 El concepto de Tipo de Datos	2 - 1
	2.2 Elementos gramaticales	2 - 2
	El conjunto de caracteres	2 - 3
	Símbolos especiales	2 - 5
	Identificadores	2 - 6
	2.3 Estructura de un programa	2 - 7
	Reglas sintácticas	2 - 8
	El encabezamiento	2 -10
	La sección de las declaraciones	2 -11
	La sección ejecutable	2 -12
	Las rutinas	2 -12
	Ambito o alcance de los identificadores	2 -13
3.	LOS TIPOS DE DATOS	3 - 1
	3.1 Los tipos ordinales	3 - 1
	3.1.1 El tipo INTEGER	3 - 2
	3.1.2 El tipo CHAR	3 - 3
	3.1.3 El tipo BOOLEAN	3 - 3
	3.1.4 Los tipos ENUMERATIVOS	3 - 3
	3.1.5 Los tipos SUBCAMPO	3 - 4
	3.2 Los tipos reales	3 - 5
	3.3 Los tipos estructurados	3 - 6
	3.3.1 El tipo ARRAY	3 - 7
	Arrays multidimensionales	3 - 9
	Cadenas de caracteres de longitud fija	3 -10

3.3.2 El tipo RECORD	3 -11
Registros con variantes	3 -13
3.3.3 Cadenas de longitud variable	3 -17
3.3.4 El tipo SET	3 -18
3.3.5 La estructura Secuencia. El tipo FILE	3 -19
Operadores elementales de ficheros	3 -21
La función EOF	3 -22
Los procedimientos READ y WRITE	3 -23
Los ficheros de texto .El tipo TEXT	3 -24
Ficheros internos y externos	3 -26
Ficheros especificados en el encabezamiento	3 -27
Asociaciones lógicas	3 -27
Asociación con fichero externo en los procedimientos RESET y REWRITE	3 -28
Otros tipos de acceso. La sentencia OPEN	3 -29
3.4 Los tipos PUNTERO	3 -30
 4. OPERADORES Y EXPRESIONES	 4 - 1
4.1 Conversiones de tipo	4 - 1
4.2 Operadores	4 - 2
4.2.1 Operadores aritméticos	4 - 2
4.2.2 Operadores relacionales	4 - 5
4.2.3 Operadores lógicos	4 - 5
4.2.4 Operadores de cadenas de caracteres	4 - 6
4.2.5 Operadores de conjuntos	4 - 6
4.3 La prioridad de los operadores	4 - 7
 5. LA SECCION DE LAS DECLARACIONES	 5 - 1
5.1 La declaración de etiquetas	5 - 1
5.2 La declaración de las constantes	5 - 2
5.3 La declaración de los tipos	5 - 2
5.4 La declaración de las variables	5 - 3

6. LAS SENTENCIAS EJECUTABLES	6 - 1
6.1 La sentencia de asignación	6 - 2
6.2 La sentencia compuesta	6 - 2
6.3 La sentencia vacía	6 - 3
6.4 Las sentencias condicionales	6 - 3
6.4.1 La sentencia IF-THEN-ELSE	6 - 3
6.4.2 La sentencia CASE	6 - 7
6.5 Las sentencias repetitivas	6 - 9
6.5.1 La sentencia WHILE	6 -10
6.5.2 La sentencia REPEAT	6 -13
6.5.3 La sentencia FOR	6 -15
6.6 La sentencia WITH	6 -17
6.7 La sentencia GOTO	6 -18
6.8 La sentencia de llamada a procedimiento	6 -18
7. LAS RUTINAS. PROCEDIMIENTOS Y FUNCIONES	7 - 1
7.1 Conceptos	7 - 2
Entidades globales, locales y estandar	7 - 3
7.2 La Declaración	7 - 5
7.3 Los parámetros. Mecanismos de sustitución	7 - 7
Parámetros-valor	7 - 9
Parámetros-variable	7 - 9
Parámetros-procedimiento y parámetros-función	7 -10
7.4 La recursividad	7 -11
El problema de las torres de Hanoi	7 -11
7.5 Las rutinas predeclaradas estandar	7 -14
Procedimientos para el manejo de ficheros	7 -14
Procedimientos de asignación dinámica de memoria	7 -15
Procedimientos de movimiento de datos	7 -15
Funciones matemáticas	7 -15
Predicados o funciones booleanas	7 -16
Funciones de transferencia entre tipos	7 -16
Otras funciones	7 -16

A.1 BIBLIOGRAFIA

## 1- INTRODUCCION

El PASCAL es un lenguaje de programación de alto nivel y de propósito general que ha derivado del ALGOL-60 y fue diseñado para enseñar técnicas de programación estructurada. Es de alto nivel porque su repertorio de instrucciones lo hacen próximo a los lenguajes humanos y a los procesos humanos de pensamiento. Sus instrucciones o sentencias se componen de expresiones de apariencia algebraica y de ciertas palabras inglesas como BEGIN, END, READ, WRITE, IF, THEN, REPEAT, WHILE, DO.

Es de propósito general como el BASIC, el COBOL, el FORTRAN, el PL/I; porque no está enfocado a un tipo específico de aplicaciones.

Pero el PASCAL, a diferencia de otros lenguajes, contiene algunos rasgos singulares que han sido diseñados para estimular el uso de la "programación estructurada", un enfoque ordenado y disciplinado de la programación que conduce a la obtención de programas claros, eficientes y libres de errores. Por ello, el PASCAL se utiliza ampliamente en la enseñanza de la informática.

Con PASCAL no sólo se dispone de un lenguaje de programación, sino que además se adquiere una metodología para el diseño y escritura de programas.

El nombre PASCAL fue elegido en honor de Blaise Pascal (1623-1662), brillante científico y matemático francés entre cuyos logros se encuentra la invención de la primera máquina de calcular mecánica.

El PASCAL fue desarrollado inicialmente a principios de los años 70 por Niklaus Wirth, en la Universidad Técnica de Zurich, Suiza. El propósito original de Wirth fue crear un lenguaje de alto nivel para enseñar programación estructurada.

La definición original del lenguaje debida a Wirth se suele conocer como PASCAL estándar o "PASCAL estándar según definición de Jensen y Wirth" (4). Pero en la actualidad el término "PASCAL estándar" resulta ambiguo porque hoy existen varios estándares diferentes.

La mayor parte de las implementaciones actuales se diferencian algo de la definición original de Wirth. La Organización Internacional de Normas (ISO/DIS 7185) ha propuesto un estándar europeo. También se desarrolla un estándar americano muy parecido bajo los auspicios conjuntos del Instituto Nacional Americano de Normas (ANSI, comité X3J9, plan BSR X3.97-1983) y del Instituto de Ingenieros Eléctricos y Electrónicos (IEEE).

El PASCAL se usa hoy ampliamente en los Estados Unidos de América y en Europa, como lenguaje de enseñanza y como lenguaje de propósito general para una gran variedad de aplicaciones diferentes. Su uso se está generalizando en ordenadores grandes y pequeños. Efectivamente el PASCAL ha llegado a ser muy popular entre usuarios de ordenadores personales, hasta el punto de que se especula sobre si podrá llegar a sustituir al BASIC como lenguaje dominante en los microprocesadores en un próximo futuro. Otros lenguajes como el FORTRAN y el COBOL, muy arraigados en los ámbitos científico y comercial, han ido adoptando algunas de las estructuras de datos y estructuras lógicas del PASCAL y son cada vez más parecidos. Es el resultado de una tendencia generalizada hacia la práctica de la programación estructurada. Hoy es frecuente oír hablar de FORTRAN estructurado, COBOL estructurado. Sin embargo, en el PASCAL es donde se encuentra la vía más natural hacia el método estructurado y, además, mantiene características diferenciadoras como la recursividad y las estructuras de datos dinámicas.

En este curso se hablará del PASCAL estándar ISO/ANSI, aunque también se presentarán algunas extensiones de uso muy común. Todo este material proporciona la base para casi todas las implementaciones comerciales del PASCAL, por lo que quien domine esta materia encontrará muy pocas dificultades en aprender otras versiones del lenguaje.

### **Obtener información a partir de datos**

Independientemente del lenguaje que se utilice, lo que se pretende casi siempre que se usa un ordenador es representar un sistema para obtener información sobre él que favorezca su conocimiento y posibilite la toma de decisiones sobre alguna materia en cuestión relacionada.

Los sistemas a tratar pueden ser de naturaleza muy diversa : la gestión administrativa de una universidad, un fenómeno físico o químico o biológico, una instalación industrial, etc... La información que interesa obtener en cada caso será diferente pero, una vez definido el problema, en todos se puede aplicar el siguiente método para su tratamiento por ordenador :

- Elegir los datos significativos.
- Organizar los datos según estructuras adecuadas a los tratamientos que se aplicarán.
- Idear algoritmos para manejar los datos que representen bien al sistema y produzcan resultados fiables.
- Escribir un programa que integre las estructuras de datos con los algoritmos y permita usar un ordenador para repetir el proceso de introducir datos y obtener los resultados correspondientes a casos diferentes.

En definitiva, hay que seguir un proceso de ascensión desde los simples datos hasta la obtención del conocimiento:

DECISIONES  
INFORMACION, CONOCIMIENTO  
PROGRAMAS  
ALGORITMOS  
ESTRUCTURAS  
DATOS

Este proceso de ascensión parece simple sobre el papel pero puede llegar a resultar muy complicado en la práctica cuando se acomete el tratamiento de sistemas reales, por eso la escritura de programas fiables no siempre se consigue en el primer intento.

Aunque la elección del lenguaje a utilizar puede parecer irrelevante cuando ya están definidas las estructuras de datos y se han construido los algoritmos; no es así especialmente en los casos donde el programa está formado por cientos de sentencias. Si no se toman precauciones, los defectos en la construcción de los programas pueden suponer una fuente continua de problemas y trastornos en el desarrollo y mantenimiento de una aplicación.

Por eso existe una tendencia general a practicar una disciplina de programación estructurada como la que facilita el PASCAL y alguna otra técnica auxiliar como el uso de diagramas estructurados. La experiencia ha permitido constatar que con el enfoque estructurado se consiguen mayores rendimientos en la programación.

### **El arte de la buena programación**

Antes de abordar el estudio del lenguaje PASCAL se van a examinar brevemente algunas características importantes de los programas bien escritos. Ello es aplicable a cualquier lenguaje, no sólo al PASCAL.

Después será conveniente utilizar este esquema como guía cuando se comience a escribir programas en PASCAL para adquirir la costumbre de hacerlo bien desde el principio.

#### **FIABILIDAD:**

Un programa ha de ser fiable ante todo. Deben estar previstos todos los casos que se puedan presentar durante su ejecución y los resultados deben ser correctos.

A veces, el programador que no ha seguido método y disciplina tiene que dedicar mucho tiempo y esfuerzo a depurar el programa hasta que se ejecuta sin errores sintácticos. Para él, el éxito ha consistido en conseguir un programa que cumple las

reglas sintácticas del lenguaje ( funciona ) y no tiene errores de escritura. Ello le ha costado un gran esfuerzo y no se detiene a comprobar si el programa funciona bien. Luego, el usuario puede sufrir las consecuencias de obtener resultados incorrectos que obligarán al programador a una depuración que puede resultar laboriosa y poco fiable por la falta de método en el diseño. En esas condiciones, es frecuente que todo el proceso deba repetirse varias veces con los consiguientes trastornos que se ocasionan al usuario.

Antes de instalar un programa, debe someterse a un conjunto exhaustivo de pruebas para garantizar cotas altas de fiabilidad.

#### EFICIENCIA:

Se suele valorar en términos de tiempo de ejecución y consumo de memoria.

El aumento de potencia de los procesadores y el abaratamiento de los componentes han permitido que los programadores trabajen con gran abundancia de medios, lo que induce a muchos a no preocuparse por la eficiencia de los algoritmos que aplican. Estas circunstancias, unidas a la divulgación alcanzada en el uso de las herramientas informáticas, están dando lugar a que la potencia de los ordenadores se desaproveche. Una parte del tiempo ejecutan cálculos inútiles por la aplicación de algoritmos ineficientes. En las comunidades de usuarios universitarios, ya es típica la figura de aquel a quien cualquier máquina se le queda pequeña, acude al Administrador del Sistema para solicitar la disponibilidad de grandes cantidades de memoria y días enteros de uso de procesador; y cuando se analiza su caso , se encuentra que tales necesidades tan desmesuradas eran consecuencia de la programación de algoritmos sumamente ineficientes.

La eficiencia de los algoritmos es un aspecto fundamental a considerar en la programación. Antes de utilizar un algoritmo debe analizarse su complejidad, que puede representarse mediante la variación del tiempo de cálculo en función de la dimensión del problema.

Para comprobar que la cuestión tiene enorme trascendencia, se puede considerar como ejemplo la comparación entre dos algoritmos para localizar un elemento en una lista ordenada. Con el método de búsqueda secuencial, es necesario recorrer toda la lista en el peor de los casos. Con el método de bisección o búsqueda binaria, se va partiendo por la mitad la lista en consideración hasta que queda vacía en el peor de los casos. En el algoritmo de búsqueda secuencial, el tiempo crece linealmente con el tamaño de la lista ( $n$ ); en el de bisección, varía según  $\log_2 n$ .

El segundo es un algoritmo de orden inferior porque crece más lentamente con " $n$ "; así que cuanto mayor sea el valor de " $n$ ", más tiempo se ahorra utilizando el segundo algoritmo. Para una lista de 50.000 elementos, con la búsqueda secuencial se necesitarían 50.000 comparaciones en el peor de los casos; mientras que con la búsqueda binaria nunca se necesitarían más de  $\log_2 (50.000)$ , que es alrededor de 16. El segundo algoritmo supone un factor de mejora de 3.000.

Pero el tema de la eficacia debe ser enfocado desde un punto de vista realista y pragmático. No sería razonable la obsesión del programador que dedica un gran esfuerzo en complicar extraordinariamente un algoritmo sólo para ahorrar algunos milisegundos en tiempo de cálculo, porque resultaría antieconómico y el producto final podría llegar a ser indescifrable para el propio autor.

Para la mayor parte de los casos, se admite que es preferible renunciar a la optimización total de los algoritmos en beneficio de la sencillez y claridad en la programación.

Sin embargo, también hay que considerar que en algunos tipos de aplicaciones, como el control de procesos en tiempo real, por ejemplo, el tiempo de ejecución puede ser crítico obligando a utilizar algoritmos de la máxima eficiencia.

#### CLARIDAD:

El programa debe estar escrito de forma que resulte legible especialmente en su lógica subyacente.

Si un programa está escrito con claridad, deberá ser posible que otro programador siga su lógica sin un esfuerzo excesivo.

Cuando un programa es poco claro, su mismo autor puede tener dificultades para entenderlo al cabo de un tiempo, hasta el punto de preferir escribirlo de nuevo si tiene que hacer alguna modificación.

Uno de los objetivos del PASCAL es favorecer la consecución de programas claros y legibles a través de un acercamiento disciplinado a la programación.

#### SIMPLICIDAD:

La fiabilidad y la claridad de los programas se potencian generalmente si se mantienen tan sencillos como sea posible.

Un programa simple es más fácil de modificar y ello adquiere importancia en la fase de mantenimiento de las aplicaciones. Las aplicaciones informáticas suelen simular sistemas reales que, son cambiantes por naturaleza. Cambian las circunstancias, los criterios, las condiciones, y es inevitable tener que modificar los programas alguna vez. En ese momento puede ocurrir que el autor no pueda dedicarse a ello porque ya no pertenezca a la organización o por estar dedicado a otro proyecto, y tenga que hacerlo otra persona. Entonces es cuando adquiere importancia la simplicidad de diseño en el programa.

En muchas ocasiones es preferible sacrificar parte de la eficacia de cálculo para mantener una estructura simple y directa. Ello no será posible en casos específicos donde se necesitan respuestas en tiempos críticos o existen condiciones muy restrictivas en la configuración del ordenador.

### MODULARIDAD:

La mayor parte de los programas largos se pueden dividir en un conjunto de subtareas identificables. Es una buena práctica implementar cada una de esas tareas como un módulo separado de programa ( en PASCAL se les llama "procedimientos" o "funciones" ).

El uso de estructuras modulares favorece la fiabilidad y claridad de los programas y facilita las modificaciones futuras.

El diseño modular conviene aplicarlo desde el principio. Un caso complejo se analiza y se va descomponiendo sucesivamente en partes más simples hasta que resulten fáciles de programar en una página de papel. Los módulos pueden ser probados por separado y ser escritos entre las diferentes personas que integran un equipo de trabajo.

### GENERALIDAD:

El valor de un programa aumenta con su rango de validez. No deberían escribirse programas válidos para un sólo caso, pero construir programas de alcance general es lo más difícil seguramente y exige una gran dedicación de recursos humanos y de tiempo.

El aumento de generalidad suele ir en contra de la simplicidad y de la eficiencia. Los programas se hacen más complejos. Conseguir programas de validez general, simples y eficientes es difícil y sólo está al alcance de personal muy cualificado.

Pero casi siempre se puede obtener una generalidad considerable y suficiente con muy poco esfuerzo de programación adicional. Por tanto conviene que ésta sea una tendencia permanente en los programadores.

### TRANSPORTABILIDAD:

Las ventajas de escribir programas transportables se reconocen en su valor cuando se sufren las consecuencias derivadas de cambiar de equipo y fabricante.

El hecho de utilizar un lenguaje de alto nivel supone una garantía de transportabilidad notable, pero no es suficiente. Hoy aún estamos lejos de la compatibilidad entre fabricantes diferentes que añaden a sus compiladores mejoras diferenciadoras sobre el estándar.

El grado de transportabilidad óptimo se consigue utilizando un compilador ajustado a normas estándar y renunciando a las extensiones de mejora introducidas por el fabricante; pero ello va en contra de la eficiencia en muchos casos.

## 2. ELEMENTOS BASICOS

Un programa en PASCAL es un conjunto de instrucciones o sentencias, escritas según ciertas reglas, para realizar operaciones sobre entidades de datos conocidas como constantes, variables y resultados de funciones.

La constante es una entidad cuyo valor no puede ser modificado durante la ejecución. La variable es una entidad cuyo valor puede modificarse durante la ejecución.

Una función es un conjunto de operaciones asociadas a un nombre y que devuelve un valor.

### 2.1. EL CONCEPTO DE TIPO DE DATOS

En PASCAL, todo dato está asociado a un TIPO de datos y debe ser declarado antes de utilizarse.

Un TIPO de datos está representado por un identificador y determina el rango de valores que un elemento de datos puede tomar, así como las operaciones a que puede ser sometido. Además, el TIPO determina el espacio en memoria necesario para almacenar cualquiera de los valores posibles que puede tomar esa entidad.

El compilador de PASCAL proporciona identificadores para algunos tipos que están predefinidos : números enteros y reales, valores lógicos, caracteres alfanuméricos, registros, tablas, cadenas de caracteres, conjuntos, ficheros y punteros a estructuras dinámicas. Pero, además, el PASCAL permite al usuario crear sus propios tipos definiendo identificadores de su elección para representar rangos de valores. Estos también llevan asociados un conjunto de operadores admisibles y unas necesidades determinadas de espacio en memoria.

El tipo de una constante es el tipo de su valor correspondiente. El tipo de una variable es el tipo establecido cuando se declaró y no puede modificarse en general. El tipo de una función es el del valor que devuelve.

Los valores de las variables y de las funciones pueden cambiar tantas veces como se desee durante la ejecución de un programa, pero esos valores deben permanecer siempre dentro del rango establecido por su tipo. Una variable no adopta un valor hasta que el programa le asigna uno. El valor de una función se calcula durante la ejecución de dicha función.

En PASCAL también existen las EXPRESIONES que, igualmente, corresponden a algún tipo siempre. Una expresión representa al valor resultante de operar alguna combinación de constantes, variables, funciones, con ciertos operadores. Se pueden usar operadores aritméticos, relacionales, lógicos, de manejo de cadenas de caracteres, y de conjuntos. Las operaciones aritméticas producen valores enteros o reales. Las operaciones relacionales y lógicas producen valores booleanos. Las operaciones entre conjuntos permiten obtener la unión, la intersección y las diferencias entre conjuntos.

Resumiendo y hablando en términos generales, se puede enunciar el principio básico de que cada constante, variable, expresión o función es de un tipo determinado.

Como en los equipos de proceso de datos sólo hay un tipo de escritura, la norma que se adopta para hacer distinciones consiste en manifestar explícitamente el tipo asociado a cada entidad en sentencias de declaración y hacer que esta declaración preceda en el texto a la utilización de las constantes, variables o funciones.

Las características del concepto de TIPO que están incorporadas al PASCAL son las siguientes :

1. Un tipo de datos determina el conjunto de valores al que pertenece una constante, o que puede tomar una variable o expresión, o que pueden ser generados por un operador o función.
2. El tipo de valor identificado por una constante, variable o expresión puede deducirse de su forma o de su declaración sin necesidad de ejecutar el proceso de cálculo.
3. Cada operador o función presupone argumentos de un tipo determinado y produce un resultado también de un tipo determinado. Si un operador admite argumentos de varios tipos ( por ejemplo, se utiliza el símbolo "+" igualmente para sumar enteros y reales ) el tipo del resultado puede determinarse a partir de reglas específicas del lenguaje.

## 2.2. ELEMENTOS GRAMATICALES

Las sentencias de un programa en PASCAL se forman con elementos que pueden ser :

- símbolos individuales, tales como los operadores aritméticos.
- palabras con un significado especial en PASCAL y palabras definidas por el usuario.

Los elementos gramaticales se construyen con caracteres. Un carácter es cualquier elemento de la tabla de codificación ASCII ( Tabla 2.1 ).

Algunos caracteres son SIMBOLOS ESPECIALES que se usan en PASCAL como delimitadores de sentencias, como operadores y como elementos sintácticos del lenguaje. Ejemplo : " ; " , " > " , " < > " , " = " . En la tabla 2.2 se indican todos.

Las PALABRAS usadas en un programa en PASCAL son combinaciones de caracteres alfabéticos y numéricos. Ocasionalmente pueden incluirse el signo del dólar (\$), el de subrayar ( \_ ) y el del tanto por ciento (%) en algunas implementaciones.

Algunas PALABRAS están RESERVADAS para nombres de sentencias ejecutables, de operaciones y de estructuras de datos predefinidas. En la tabla 2.3 se presentan todas las PALABRAS RESERVADAS. Ejemplo : ARRAY, BEGIN, CASE, DIV, etc...

Otras PALABRAS usadas en los programas son los IDENTIFICADORES. Algunos están PREDEFINIDOS y representan rutinas y tipos de datos que proporciona el PASCAL. Otros identificadores usados son los DEFINIDOS POR EL USUARIO para poner nombre a programas, constantes simbólicas, variables y cualquier elemento del programa que no haya sido denominado. En las tablas 2.4 y 2.5 se indican los IDENTIFICADORES PREDEFINIDOS. Ejemplo : CHAR, EOF, EXP, WRITE, etc...

### **El conjunto de caracteres**

El PASCAL estandar usa el conjunto de la tabla de codificación ASCII ( American Standard Code for Information Interchange ) que contiene 128 caracteres puestos en orden.

Hay varias clases de caracteres :

- . Las letras mayúsculas y las minúsculas desde la " A " a la " Z " y desde la " a " a la " z " .
- . los números del " 0 " al " 9 " .
- . Caracteres especiales, tales como el "ampersand" (&), el signo de interrogación (?), el signo de igualdad (=).
- . Caracteres no imprimibles, tales como el espacio en blanco, el tabulador, el salto de línea, el retorno de carro, el pitido.

En general, los compiladores de PASCAL no distinguen entre letras mayúsculas y minúsculas excepto cuando se escriben entre apóstrofes.

Ejemplo : PROGRAM Uno y PROGRAM UNO son equivalentes.  
' Pepe Pérez ' y ' PEPE PEREZ ' son diferentes.

La tabla 2.1 resume el conjunto de caracteres ASCII. Cada elemento es un valor constante del tipo predefinido CHAR.

El número decimal que lleva asociado cada elemento es el ordinal que devuelve la función ORD de PASCAL aplicada a ese carácter.

Izda.	Derecha									
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

**Tabla 2.1**  
**Conjunto de caracteres ASCII**

Significado de algunas abreviaturas:

nul	nulo	nl	salto de linea
ht	tabulador horizontal	esc	escape
cr	retorno de carro	bs	retroceso
bel	campana	vs	tabulador vertical

Observaciones:

- Los códigos de los caracteres 0 a 31 y 127 no son imprimibles
- Los códigos de las letras mayúsculas, los de las minúsculas y los de las cifras son contiguos entre sí.
- La diferencia entre una letra mayúscula y su correspondiente minúscula es 32.

### Símbolos especiales

Se usan para representar delimitadores, operadores y otros elementos sintácticos.

Cuando un símbolo está formado por más de un carácter, éstos deben escribirse seguidos sin espacios en blanco de separación.

Nombre	Símbolo	Nombre	Símbolo
Sumar, signo más	+	Menor que	<
Restar, signo menos	-	Menor o igual que	<=
Multiplicar	*	Mayor que	>
Dividir	/	Mayor o igual que	>=
Operador de asignación	:=	Paréntesis	( )
Punto	.	Corchetes	[ ]
Punto y coma	;	Comentarios	{ }, (* *)
Dos puntos	:	Puntero	@, ^
Apóstrofo	'	Operador de subrango	..
Igual	=	Distinto	<>

**Tabla 2.2**  
**Símbolos especiales en el PASCAL**

### Palabras reservadas

En la definición del lenguaje PASCAL, algunas palabras están reservadas. Se usan como nombres de sentencia, de tipos de datos y de operadores. Suelen escribirse con mayúsculas, aunque no es necesario.

Las palabras reservadas sólo se pueden usar, dentro de un programa, en el contexto para el que han sido definidas. No se pueden redefinir para usarlas como identificadores.

La tabla 2.3 muestra las palabras reservadas estándar.

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
BEGIN	FOR	OF	TO
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNTO	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

**Tabla 2.3**  
**Palabras reservadas estándar**

### Identificadores

Son nombres que denotan constantes, tipos, variables, procedimientos y funciones. Pueden incluir letras y dígitos cumpliendo las restricciones siguientes:

- . Un identificador no puede comenzar con un dígito.
- . Un identificador no puede tener espacios en blanco ni dígitos especiales.
- . Los primeros caracteres de un identificador (31 en VAX PASCAL) deben designar un nombre único dentro del bloque de programa en el que ha sido definido.

En PASCAL hay algunos identificadores que están predeclarados como nombres de procedimientos, de funciones, tipos de datos, constantes simbólicas y variables de fichero. En la tabla 2.4 se indican los incluidos en el estándar.

ABS	FALSE	PACK	SQR
ARCTAN	GET	PAGE	SQRT
BOOLEAN	INPUT	PRED	SUCC
CHAR	INTEGER	PUT	TEXT
CHR	LN	READ	TRUE
COS	MAXINT	READLN	TRUNC
DISPOSE	NEW	REAL	UNPACK
EOF	NIL	RESET	WRITE
EOLN	ODD	REWRITE	WRITELN
EXP	ORD	ROUND	
	OUTPUT	SIN	

**Tabla 2.4**  
**Identificadores predeclarados estándar**

En un programa puede redefinirse cualquier identificador predeclarado para denominar otra entidad. Si se hace eso, no podrá utilizarse tal identificador para su propósito habitual dentro del bloque donde se haya redefinido.

No es recomendable redefinir identificadores predeclarados porque se pierde acceso a prestaciones del lenguaje útiles.

Los identificadores definidos per el usuario se usan para nombrar programas, módulos, constantes, variables, procedimientos, funciones, secciones de un programa y tipos definidos por el usuario. Estos identificadores representan estructuras de datos significativas, valores y acciones que no están representadas por alguna palabra reservada, identificador predeclarado o símbolo especial.

### 2.3. ESTRUCTURA DE UN PROGRAMA

Un programa escrito en PASCAL consta de un ENCABEZAMIENTO y un BLOQUE.

En el encabezamiento se especifica el nombre del programa y los nombres de los ficheros externos que se usan para entrada de datos y salida de resultados.

El bloque está dividido en dos partes :

- . La sección de las declaraciones, donde se declaran todos los datos y las rutinas.
- . La sección ejecutable, que contiene sentencias ejecutables.

Un programa también puede contener líneas de comentario intercaladas en cualquier lugar.

Sintácticamente, un programa comienza con la palabra PROGRAM y termina con un punto "."

El primer ejemplo muestra un caso sencillo donde sólo se pretende ofrecer un visión general :

```
PROGRAM Uno ( INPUT, OUTPUT ) ; (* Este es el encabezamiento *)
(* Ejemplo PAS001 *)
(* Aquí no hay declaraciones. No se necesitan *)
BEGIN
    WRITELN ( ' Que tal va con este capítulo ? ' ) ; (* Esta es la única sentencia
END.
```

El resultado de su ejecución consiste en escribir por la terminal el texto siguiente:

Que tal va con este capítulo ?

### Reglas sintácticas

1. El punto y coma ( ; ) y el punto ( . ) son DELIMITADORES en PASCAL.

El punto y coma separa sentencias consecutivas. También se usa para terminar el encabezamiento del programa y las declaraciones de los datos. No es necesario escribir punto y coma después de la palabra BEGIN ni antes de la palabra END porque BEGIN y END no son sentencias.

El punto indica el final del programa.

2. Las palabras BEGIN y END también son delimitadores, no son sentencias. Se usan para separar las partes funcionales de un programa. Con ellas se indica el principio y el final de la sección ejecutable. También sirven para delimitar una sentencia compuesta. Cada BEGIN debe estar asociado con un END, excepto en dos casos : la sentencia CASE y la declaración de RECORD.
3. El PASCAL permite formato libre en la escritura del texto que compone el programa. Se pueden colocar las sentencias en cualquier lugar de una línea, escribir una sentencia en más de una línea y colocar varias sentencias en una misma línea. Pero no se puede dividir un nombre y un número entre varias líneas o con un espacio en blanco.
4. Un programa puede contener comentarios en cualquier lugar. Los comentarios se delimitan encerrándolos entre llaves ( { } ). También es posible comenzar un comentario con " ( \* " y acabarlo con " \* ) ".

En el segundo ejemplo que se muestra a continuación, se presenta un programa algo más completo para indicar el aspecto de las partes que lo componen. En este momento no hay que prestar demasiada atención a las sentencias que contiene porque se estudiarán más adelante. El programa simula el comportamiento de un transeunte cuando se dispone a cruzar un paso de peatones con semáforo. Evidentemente es demasiado simple.

```

PROGRAM Cruzar_calle ( INPUT, OUTPUT ) ;
(* Ejemplo PAS002 , en VAX-PASCAL *)
(* *)
(* Sección de las declaraciones *)
TYPE (* Declaración del tipo "colores" *)
    colores = (verde,rojo,ámbar,amarillo) ;
(* *)
VAR color : colores ; (* Declaración de la variable "color" *)
(* *)
(* No hay más declaraciones *)
(* *)
BEGIN (* Aquí comienza la parte ejecutable *)
    WRITELN ( ' De que color está el semáforo (verde, rojo, ámbar) ? ' ) ;
    READLN (color) ; (* en PASCAL estandar, es errónea *)
    CASE color OF
        verde : WRITELN(' PUEDES PASAR, PERO MIRA POR SI ACASO ');
        amarillo, ámbar, rojo : WRITELN ( ' QUIETO !, ESPERA ' )
    END ;
END.

```

Este mismo programa sin tanto comentario es así:

```

PROGRAM Cruzar_calle ( INPUT, OUTPUT ) ;
TYPE
    colores = (verde,rojo,ámbar,amarillo) ;
VAR
    color : colores ;
BEGIN
    WRITELN ( ' De que color está el semáforo (verde, rojo, ámbar) ? ' ) ;
    READLN (color) ;
    CASE color OF
        verde : WRITELN(' PUEDES PASAR, PERO MIRA POR SI ACASO ');
        amarillo, ámbar, rojo : WRITELN ( ' QUIETO !, ESPERA ' )
    END ;
END.

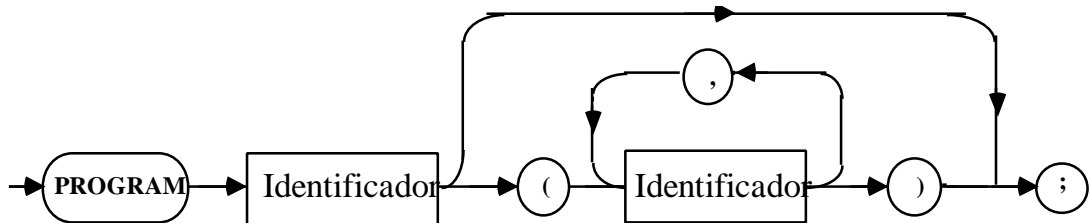
```

### El encabezamiento

Un programa escrito en PASCAL empieza siempre con un encabezamiento que consiste en :

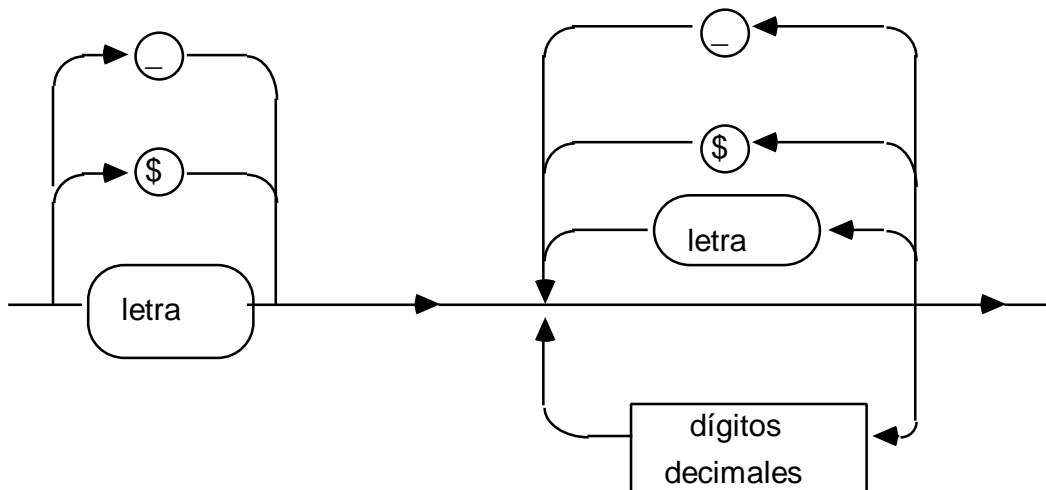
- . La palabra reservada PROGRAM.
- . El nombre del programa.
- . Los nombres de los ficheros externos que se usan para entrada y salida. Se escriben separados por comas y entre paréntesis.
- . El delimitador punto y coma.

En un diagrama de Conway, se representa así:

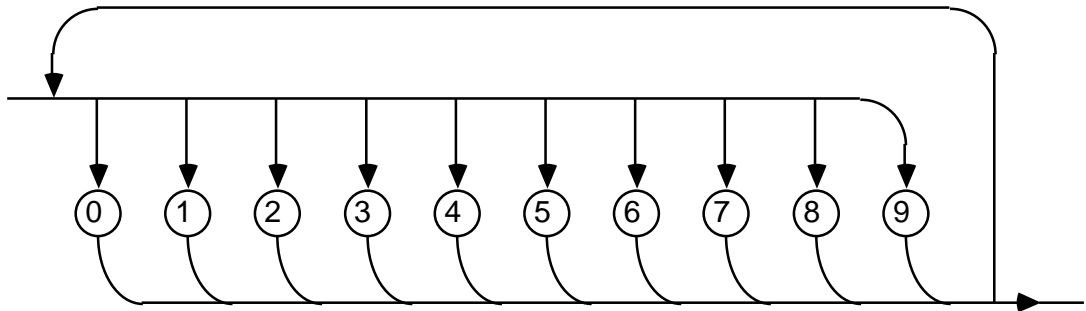


En este tipo de diagramas, muy usados para describir la sintaxis en PASCAL, lo que está dentro de círculos o de óvalos debe aparecer exactamente tal como se muestra. Lo que está dentro de rectángulos, se describe con otro diagrama.

El identificador se representa así :



Para representar los dígitos decimales, se puede usar el diagrama siguiente :



Ejemplos de encabezamientos válidos :

```
PROGRAM Uno :
PROGRAM Dos ( INPUT, OUTPUT ) ;
PROGRAM Tres ( INPUT, OUTPUT, AUXILIAR ) ;
```

### La sección de las declaraciones

Todas las entidades definidas por el usuario que se usan en un programa deben ser declaradas en esta sección, indicando un identificador y lo que representa.

ETIQUETAS	( LABEL )
CONSTANTES	( CONST )
TIPOS	( TYPE )
VARIABLES	( VAR )
PROCEDIMIENTOS	( PROCEDURE )
FUNCIONES	( FUNCTION )

No es necesario que un programa contenga declaraciones de todas esas categorías. Podría no haber ninguna como en el ejemplo PAS001.

Las declaraciones pueden escribirse en un orden cualquiera.

Una misma clase de declaración puede aparecer más de una vez. Pero una declaración particular no se puede repetir en un bloque.

Las etiquetas son enteros decimales que pueden usarse para señalar alguna sentencia y hacerla accesible mediante la sentencia GOTO.

Aquí ya no se va a dedicar más atención a las sentencias de declaración porque se describirán más adelante, pero se indican algunos ejemplos para mostrar la forma que adoptan:

```
TYPE estaciones = (primavera, verano, otoño, invierno) ;
VAR letra : CHAR;
CONST pi = 3.1416 ;
LABEL 100, 200, 300 ;
```

### **La sección ejecutable**

Contiene a las sentencias que, cuando se ejecutan, realizan las acciones del programa.

La sección ejecutable sigue a la de las declaraciones y está delimitada por BEGIN y END, terminando en un punto.

Entre BEGIN y END puede haber sentencias condicionales, repetitivas, sentencias de asignación, sentencias que controlan el flujo de la ejecución.

Tampoco se va a dedicar más atención a ésta cuestión aquí porque más adelante se describirán todas esas clases de sentencias.

### **Las Rutinas**

El PASCAL permite agrupar definiciones, declaraciones y sentencias ejecutables en RUTINAS.

Es muy conveniente usar rutinas para organizar un programa aislando las tareas particulares que comporta. Practicando esta técnica desde el principio se puede conseguir un diseño modular de los programas que favorece el refinamiento por etapas sucesivas durante la etapa de desarrollo y facilita el mantenimiento posterior.

En PASCAL se usan dos tipos de rutinas: los PROCEDIMIENTOS y las FUNCIONES.

Los procedimientos se usan para realizar un conjunto de acciones. Se invocan con una sentencia ejecutable denominada " llamada a procedimiento "

Las funciones se usan para obtener y devolver un valor. Se invocan cuando un identificador de función aparece dentro de una expresión.

El compilador de PASCAL suministra muchas rutinas predeclaradas para realizar operaciones de uso frecuente, como las de entrada y salida.

Una rutina consta de un encabezamiento y un bloque que hay que definir en la declaración de la rutina ( recordar que los procedimientos y funciones son entidades que hay que declarar en los programas ). El encabezamiento contiene el nombre de la rutina, una lista de parámetros formales que declaran los datos externos a la rutina y, en el caso de las funciones, el tipo del resultado o valor de la función.

El bloque de la rutina contiene una sección de declaraciones opcional y una sección ejecutable. En la primera se declaran datos que son locales a la rutina y son inaccesibles o desconocidos desde fuera.

### **Ambito o alcance de los identificadores**

El PASCAL es un lenguaje para la programación estructurada en bloques. Permite anidar bloques de rutinas, no sólo dentro del programa principal, sino también dentro de otra rutinas.

Cada rutina puede tener sus propias declaraciones y definiciones locales, incluso se puede redeclarar en ella un identificador que haya sido declarado en un bloque más externo.

Una rutina declarada en un nivel más interno tiene acceso a las declaraciones y definiciones hechas en todos los bloques que la contienen.

La parte del programa donde se tiene acceso a un identificador se llama ámbito o alcance del identificador. Fuera de ese dominio, un identificador no tiene significado o tiene significado diferente.

Concretamente, **el ámbito de un identificador es el bloque donde ha sido declarado.** Como los bloques pueden estar anidados, el alcance de un identificador puede incluir a bloques situados a niveles inferiores en la jerarquía del programa.

En el ejemplo del diagrama que sigue es fácil comprobar las siguientes afirmaciones

- . Una variable declarada en "Chequear" sólo es conocida allí.
- . Una variable declarada en "Altas" es conocida en "Altas", "Leer", "Chequear", "Mostrar", "Ingresar"; pero no es conocida en las demás.
- . Una variable declarada en "Actualizar" es conocida en todos los demás procedimientos.

PROGRAM Actualizar

PROCEDURE Menu

PROCEDURE Leer\_opcion

PROCEDURE Ayuda

PROCEDURE Altas

PROCEDURE Leer

PROCEDURE Chequear

PROCEDURE Mostrar

PROCEDURE Ingresar

En el tercer ejemplo ( PAS003 ), que se expone seguidamente, se avanza presentando un programa un poco más completo que los anteriores para que el lector pueda contemplar el aspecto que tienen las secciones de un programa. Allí hay declaraciones de un tipo, de variables y de una función; y aparecen dos sentencias importantes " IF - THEN - ELSE " y " CASE ". Es el programa " Despertador " que puede constituir una aproximación al discurso en el cerebro de un ciudadano somnoliento poco después de que suena el despertador por la mañana.

```

PROGRAM Despertador ( INPUT, OUTPUT ) ;      ( 1 )
(* Ejemplo PAS003 , en VAX-PASCAL *)
(**)
TYPE          ( 2 )
    dias_de_la_semana = (lunes,martes,miercoles,jueves,
                        viernes, sabado, domingo) ;

(**)
VAR           ( 3 )
    dia : dias_de_la_semana ;
    llueve, hace_viento : CHAR ;

(**)
FUNCTION Hace_buen_dia : BOOLEAN ; ( 4 )
BEGIN
    IF (llueve = 'N') AND (hace_viento = 'N') THEN
        hace_buen_dia := TRUE
    ELSE
        hace_buen_dia := FALSE ;
END ;
(**)
BEGIN          ( 5 )
    WRITELN ( ' RIIING !! ' ) ;
    WRITELN ( ' Que dia es hoy ? ' ) ;
    READLN (dia) ; (* en PASCAL estandar, es errónea *)
    CASE dia OF
        lunes, martes, miercoles, jueves, viernes:
            WRITELN ( ' LEVANTATE, HAY QUE IR A TRABAJAR ' ) ;
        sabado, domingo :
            BEGIN          ( 6 )
                WRITELN ( ' Llueve (S/N) ? ' ) ;
                READLN (llueve) ;
                WRITELN ( ' Hace viento (S/N) ? ' ) ;
                READLN (hace_viento) ;
                IF hace_buen_dia THEN      ( 7 )
                    WRITELN ( ' LEVANTATE Y A DISFRUTAR DEL DIA,
                ELSE
                    WRITELN( ' TRANQUILO, PUEDES SEGUIR TUMBADO' )
            END ;
    END ;
END .

```

En primer lugar merece la pena observar el aspecto general del programa y fijarse en los sangrados escalonados que se usan al escribir las sentencias para facilitar la lectura y análisis del programa.

Notar que se usan identificadores cuyos nombres aluden directamente al significado de la variable que representan. Esto facilita mucho la comprensión del programa.

En (1) está el encabezamiento. Se ve que contiene, como de costumbre, a los identificadores INPUT y OUTPUT. Son dos identificadores predefinidos en PASCAL que corresponden a los ficheros para la entrada y salida usados por defecto. En cualquier instalación suelen quedar asignados a los dispositivos estándar para la entrada y la salida que están identificados con la video-terminal usada para el trabajo en régimen interactivo. Después, con READLN se va a leer por INPUT y con WRITELN se escribe por OUTPUT.

En (2), (3) y (4) están las declaraciones que aquí incluyen a un tipo, algunas variables y una función. El tipo " dias\_de\_la\_semana " es de los definidos por el usuario, no está predeclarado y por eso hay que definirlo. Los tipos CHAR y BOOLEAN son de los predefinidos en PASCAL.

Notar que la declaración de la función incluye su definición completa.

En (5) comienza la sección ejecutable que, básicamente está formada por una sentencia CASE. Notar que en el caso de que el día sea " sábado " o " domingo ", se ejecuta una sentencia compuesta (6) delimitada por BEGIN y END.

En la definición de la función, que comienza en (4), hay dos sentencias de asignación. Notar que el símbolo para la asignación es " := ".

### 3. LOS TIPOS DE DATOS

En PASCAL hay cuatro categorías de tipos de datos:

Ordinales:	INTEGER	( números enteros )
	CHAR	( caracteres )
	BOOLEAN	( valores lógicos : TRUE y FALSE )
	Enumerativos Subcampo	
Reales:	REAL	( números reales )
Estructurados:	ARRAY	( tablas )
	RECORD	( registros )
	FILE	( secuencias )
	SET	( conjuntos )

Punteros.

A los tipos Ordinales y Reales se les llama habitualmente tipos escalares o tipos simples. Son los tipos fundamentales que sirven para construir tipos estructurados.

Los tipos INTEGER, CHAR, BOOLEAN y REAL están predefinidos por el compilador. Los tipos Enumerativos y Subcampo son definidos por el usuario.

Los tipos estructurados permiten procesar grupos de datos ordinales, reales, estructurados y punteros. Por ejemplo, se pueden tener una cadena de caracteres, un fichero de registros, una tabla de punteros. Todas son estructuras estáticas.

El tipo puntero permite manejar estructuras de datos dinámicas. Sus valores son direcciones de almacenamiento de variables dinámicas.

#### 3.1. LOS TIPOS ORDINALES

Los valores de un tipo ordinal tienen una correspondencia biunívoca con el conjunto de enteros positivos. Tales valores están ordenados de manera que a cada uno le corresponde un valor ordinal único que indica su posición en la lista de todos los valores posibles de ese tipo.

Hay tres funciones predeclaradas que operan solamente sobre expresiones de tipo ordinal, proporcionando información sobre la secuencia de valores ordenados de ese tipo:

- . La función PRED busca al predecesor de cualquier valor de un tipo ordinal, excepto para el menor de todos.
- . La función SUCC busca al sucesor de cualquier valor de un tipo ordinal, excepto para el mayor de todos.
- . La función ORD busca al ordinal de un valor y lo devuelve como un entero. El ordinal de un entero es el mismo entero.

### 3.1.1. El tipo INTEGER

Incluye a los valores enteros positivos y negativos desde  $-2^{31}+1$  hasta  $2^{31}-1$ . Este campo contiene valores que van desde -2.147.483.647 hasta 2.147.483.647. Ocupa 32 bits en memoria.

El mayor valor posible de tipo INTEGER está asignado al identificador de constante predefinido MAXINT.

Los valores de tipo INTEGER se representan con dígitos decimales. No se admiten ni el punto ni la coma. Estos son valores correctos:

```
32
0
87314
```

También se pueden usar enteros negativos pero hay que tener en cuenta que un entero negativo como -32, por ejemplo, no es una constante, es una expresión formada por el signo (-) y el valor entero 32. El uso de enteros negativos en expresiones aritméticas complejas puede dar lugar a resultados distintos de los esperados si no se interpreta bien esta cuestión. Más adelante se insistirá sobre este tema.

Las operaciones de entrada (lectura) permiten suministrar el signo "más" o "menos" con los valores enteros. En las operaciones de salida, el signo de los valores negativos se incluye automáticamente.

Los datos de tipo INTEGER pueden ser sometidos a operaciones aritméticas y otras que se explican en el capítulo 4.

### 3.1.2. El tipo CHAR

Sus valores posibles son cada uno de los elementos del conjunto de caracteres ASCII. Ocupan ocho bits en memoria.

Para especificar un caracter constante basta con escribir cualquier caracter ASCII imprimible entre apóstrofos. Para especificar el apóstrofo mismo hay que escribirlo dos veces entre apóstrofos. Cada una de las siguientes es una constante de tipo CHAR válida:

'A'	'.'
'Z'	'"' (es el apóstrofo)
'0'	'?'

Se pueden escribir cadenas de caracteres como 'BUENOS DIAS' y 'que tal está?', pero deben ser tratados como datos estructurados tales como las tablas ( ARRAY ) de caracteres.

Al aplicar la función ORD sobre una expresión de tipo CHAR, se obtiene como resultado el valor ordinal que indica su posición en la tabla ASCII. Por ejemplo, si la variable "letra\_A" tiene el valor 'A', entonces la expresión ORD (letra\_A) devuelve el valor 65 (consultar la tabla).

Notar que en la tabla ASCII, cada uno de los conjuntos de dígitos, las letras mayúsculas y las letras minúsculas ocupan posiciones consecutivas y que las letras mayúsculas están antes que las minúsculas.

Las variables de tipo CHAR pueden ser sometidas a operaciones de concatenación y comparación tal como se explica en el capítulo 4.

### 3.1.3. El tipo BOOLEAN

Los datos de tipo BOOLEAN sólo pueden tomar dos valores constantes que se describen con los identificadores predeclarados FALSE (falso) y TRUE (verdadero). Tales valores están ordenados de forma que FALSE es menor que TRUE.

La función ORD aplicada sobre el valor FALSE devuelve el entero 0 y aplicada sobre TRUE devuelve el entero 1.

Al chequear la veracidad o validez de las relaciones de comparación se obtiene como resultado un valor de tipo BOOLEAN.

### 3.1.4. Los tipos ENUMERATIVOS.

Un tipo Enumerativo es un conjunto de valores constantes ordenados representados por identificadores.

Estos tipos no están predefinidos por el compilador. Los inventa el usuario.

En la declaración es donde el usuario define el tipo listando todos los identificadores en orden, separados por comas y encerrados entre paréntesis.

Sintaxis: ( { identificador }, ... )

Ejemplo: ( Primavera, Verano, Otoño, Invierno )

Los valores de un tipo enumerativo quedan ordenados según su posición en la lista de izquierda a derecha. A cada identificador le queda asignado un valor ordinal comenzando en cero para el primero.

La función ORD puede aplicarse sobre expresiones de tipo enumerativo. Para el ejemplo propuesto, las siguientes afirmaciones son correctas:

ORD (Primavera) es menor que ORD (Verano)

ORD (Verano) es igual a 1

Los identificadores de los tipos enumerativos no pueden ser definidos para otros propósitos dentro del mismo bloque de programa.

Con el PASCAL para VAX-11, un tipo enumerativo puede tener hasta 65535 identificadores como máximo.

Otros ejemplos de estos tipos:

(agua, vino, cerveza, leche)

(cilindro, cono, esfera)

(recluta, soldado, cabo, sargento)

(fluor, cloro, bromo, yodo)

### 3.1.5. Los tipos SUBCAMPO

Con un tipo Subcampo, o Subrango, se especifica una parte limitada de otro tipo ordinal (llamado tipo base) para ser usado como un tipo distinto. Es el usuario quien lo crea.

En la sentencia donde se declara, se indican los límites que lo definen:

Sintaxis: límite inferior .. límite superior

límite inferior: expresión constante que establece el límite inferior del subcampo.

límite superior: expresión constante que establece el límite superior del subcampo.

Ejemplos: '0' .. '9'  
'A' .. 'M'  
1 .. 31  
Enero .. Marzo

Un tipo subcampo está definido sólo para los límites y los valores comprendidos entre ellos.

El valor del límite superior debe ser mayor o igual al del límite inferior.

Con el símbolo de subcampo ( .. ) se separan los límites.

El tipo base puede ser cualquier tipo ordinal predefinido o enumerativo. Los valores en el tipo subcampo están en el mismo orden que en el tipo base. Así, el resultado de la función ORD aplicada a un valor de un tipo subcampo es el ordinal que aquel valor tiene asociado por su posición relativa en el tipo base, no en el tipo subcampo.

Un tipo subcampo se puede usar en cualquier lugar de un programa donde su tipo base esté definido. Un valor de tipo subcampo se convierte a un valor de su tipo base antes de ser usado en cualquier operación. Todas las reglas que gobiernan las operaciones con un tipo ordinal son aplicables a los subcampos de este tipo.

El uso de los tipos subcampo puede mejorar la claridad de los programas. Por ejemplo, se pueden limitar los valores admisibles para los días del año definiendo el tipo 1..366.

### 3.2. Los tipos REALES

El tipo REAL es el estándar para representar valores numéricos reales. El PASCAL para VAX-11 proporciona algunos otros que permiten manejar valores reales en un rango más amplio con distinto grado de precisión. Básicamente son el tipo DOUBLE, para precisión doble; y el tipo QUADRUPLE, para precisión cuádruple.

En la tabla 3.1 se indican los rangos de valores y los grados de precisión para estos tipos de datos.

Los números reales pueden escribirse en notación decimal o en notación exponencial. En la decimal se usan el conjunto de los dígitos decimales y el punto. Siempre debe aparecer algún dígito en cualquiera de los lados del punto.

Son ejemplos correctos:	3.14
	976.251
	7.0
	0.0
Son incorrectos:	7.
	.14

Para representar números muy grandes o muy pequeños es más conveniente utilizar la notación exponencial (o de coma flotante) que se construye con un número real o uno entero, una letra para indicar el grado de precisión (E, D, Q) y un entero que representa al exponente y puede llevar signo.

Menor valor negativo	-0.29E-38	-0.29D-38	-0.84Q-4932
Mayor valor negativo	-1.70E38	-1.70D38	-0.59Q4932
Menor valor positivo	0.29E-38	0.29D-38	0.84Q-4932
Mayor valor positivo	1.70E38	1.70D38	0.59Q4932
Precisión	1 parte en $2^{23}$ = 7 dígitos decimales	1 parte en $2^{55}$ = 16 dígitos decimales	1 parte en $2^{112}$ = 33 dígitos decimales

**Tabla 3.1**  
**Rangos y precisión para los tipos REAL, DOUBLE, QUADRUPE en VAX-PASCAL**

Son ejemplos correctos:

2.3e2  
10.0E-1  
9.1416E0  
2370E-6

Con los valores reales negativos hay que repetir la advertencia que se hizo para los enteros. Un valor tal como  $-4.5E+3$  no es una constante, es una expresión donde el signo (-) es el operador; y esta cuestión obliga a ser cuidadoso al construir expresiones aritméticas complejas para que sean evaluadas de la forma deseada.

Respecto al tratamiento del signo en las operaciones de entrada y salida, pasa lo mismo que con los enteros. En la entrada se admite la especificación del signo. En la salida se escribe automáticamente en los valores negativos.

### 3.3.Los tipos ESTRUCTURADOS.

Los tipos estructurados pueden contener más de un componente a la vez, a diferencia de los tipos ordinales y reales. Cada componente puede ser de tipo ordinal, real, estructurado o puntero. Se puede tener acceso a cada uno de los componentes o procesar la estructura completa.

Los tipos estructurados se caracterizan por el tipo o tipos de sus componentes y por la manera en que están organizados los mismos.

Las estructuras fundamentales son cuatro:

ARRAY	(tabla)
RECORD	(registro)
SET	(conjunto)
FILE	(secuencia)

En el PASCAL para VAX-11 también existe la estructura VARYING OF CHAR para manejar cadenas de caracteres de longitud variable.

Para cada tipo estructurado, excepto FILE, se pueden expresar valores constantes formando constructores. Un constructor de ARRAY o de RECORD debe contener un valor constante de tipo apropiado para cada componente de la estructura. Se pueden usar constructores con varios propósitos:

- . Para definir constantes simbólicas en la sección CONS
- . Para inicializar variables de tipo estructurado en la sección VAR.
- . Para asignar valores a variables estructuradas en la sección ejecutable.
- . Para pasar parámetros a las rutinas.

El operador inverso al constructor es el selector, que constituye la manera de identificar componentes particulares de la estructura. Tiene distinta forma para cada tipo.

Para ahorrar espacio de almacenamiento en memoria, se pueden empaquetar los objetos de cualquiera de los tipos estructurados mencionados. Para crear tipos estructurados empaquetados, basta con indicar la palabra PACKED delante de la definición del tipo. Las estructuras son almacenadas en el menor número de bits posible.

### 3.3.1. El tipo ARRAY

La organización "array" es, probablemente, la estructura de datos más conocida debido a que en muchos lenguajes es la única disponible explícitamente.

Un "array" es una estructura homogénea que está constituida por componentes del mismo tipo, llamado tipo base.

También se denomina estructura de acceso aleatorio, o acceso directo, porque todos sus componentes pueden seleccionarse arbitrariamente y son igualmente accesibles. Para designar a un componente aislado, el nombre de la estructura se amplía con el llamado índice de selección del componente. El índice debe ser un valor del tipo definido como tipo índice del array.

La definición de un tipo "array" T especifica un tipo base  $T_0$  y un tipo índice I.

```
TYPE T = ARRAY [ I ] OF T0
```

Ejemplos:

```
TYPE vector = ARRAY [1..3] OF REAL
TYPE linea = ARRAY [1..80] OF CHAR
TYPE nombre = ARRAY [1..32] OF CHAR
```

Los índices de un "array" deben ser de tipo ordinal. No suele ser posible utilizar el tipo INTEGER porque se excedería el espacio de memoria accesible. Para usar valores enteros como índices, hay que indicar un subrango de enteros.

Un valor de tipo "array" se puede designar por un constructor donde se indica el nombre del tipo y valores constantes de los componentes separados por comas y encerrados entre paréntesis.

Ejemplo: vector (2.57, 3.14, -8.16)

Para especificar un mismo valor a componentes consecutivos, se puede usar un factor de repetición de la forma "n of valor"; donde "n" indica el número de componentes que recibirán el mismo valor, y debe ser una expresión constante de tipo INTEGER. El "valor" puede ser una constante u otro constructor del mismo tipo que el componente.

Ejemplos: vector (3 OF 1.0) es equivalente a vector (1.0,1.0,1.0)  
vector (7.2, 2 OF 3.0)

Para seleccionar un componente individual de un "array" se usa un selector que consiste en escribir el nombre de la variable de tipo "array" seguido por el nombre correspondiente al componente, escrito entre corchetes.

Ejemplo: si v := vector (2.57, 3.14, -8.16)  
entonces v [1] es igual a 2.57.

Los componentes de un "array" se pueden someter a todas las operaciones permitidas para su tipo. La única operación definida para el "array" como tal, es la asignación ( := ).

La forma habitual de operar con "arrays" es actualizar componentes aislados.

El hecho de que los índices tengan que ser de un tipo de datos definido tiene una consecuencia importante: los índices pueden calcularse, es decir, puede ponerse una expresión en lugar de una constante. La expresión debe evaluarse y su resultado determina el componente seleccionado. Esta generalización proporciona una herramienta de programación potente pero, al mismo tiempo, da ocasión a uno de los errores de programación más frecuentes: el valor resultante puede estar fuera del intervalo especificado como campo de variación de los índices. Se supone que en tales casos, la implementación en un equipo concreto ha previsto la emisión de mensajes de error.

Los valores organizados en forma de "array" se suelen manejar utilizando índices variables y sentencias repetitivas que permiten construir algoritmos para realizar tratamientos sistemáticos.

Los índices son de tipo escalar, es decir, un tipo no estructurado en el que está definida una relación de orden. Si el tipo base también está ordenado, entonces queda establecida una relación de orden para ese tipo "array".

La relación de orden entre dos "arrays" está determinada por los dos componentes desiguales, correspondientes, con menor índice.

### "Arrays" multidimensionales

Los componentes de un "array" pueden estar estructurados. Una variable de tipo "array" cuyos componentes también son "arrays" se llama matriz.

Un ejemplo de "array" bidimensional:

```
TYPE fila = ARRAY [1..3] OF REAL
      matriz = ARRAY [1..3] OF FILA
```

Esto es equivalente a:

```
matriz = ARRAY [1..3] OF ARRAY [1..3] OF REAL
```

o bien:

```
matriz = ARRAY [1..3, 1..3] OF REAL
```

Un "array" puede tener un número cualquiera de dimensiones y cada dimensión puede tener índice de tipo diferente.

Ejemplo:

```
TYPE
      enteros = ARRAY [0..10] OF ARRAY ['A'..'D'] OF INTEGER
      enteros = ARRAY [0..10, 'A'..'D'] OF INTEGER
```

Cada elemento de un "array" multidimensional se describe indicando, entre corchetes, valores para todos los índices en el orden en que fueron declarados:

Ejemplo:

```
m [1,2] , entero [1,'B']
si "m" es de tipo "matriz" y "entero" es de tipo "enteros".
```

El constructor de un "array" multidimensional contiene constructores como componentes.

Ejemplos:

- para el tipo: `tabla = ARRAY [0..3,1..5] OF INTEGER`

un constructor posible es:

`((2, 5, 7, 1, 0), 2 OF (5 OF 1), (7, 2,3 OF 5))`

- para el tipo:

`matriz_3_D = ARRAY[1..3] OF ARRAY[1..4] OF ARRAY[1..2] OF INTEGER`

un constructor posible es:

`(( (2,4), (1,2), (7,5), (1,-2)) ,  
 ((3,7), (-2,3), (5,6), (9,8)) ,  
 ((-3,-1), (1,2), (7,5), (6,1)))`

Para todas las dimensiones, excepto la más interna, los valores resultantes se escriben en forma de constructores porque sus componentes son de tipo "array".

### **Cadenas de caracteres de longitud fija**

En PASCAL, se definen como "array" empaquetado de caracteres con límite inferior de índice igual a 1. La longitud de la cadena queda establecida por el límite superior.

Ejemplo: `TYPE nombre = ARRAY [1..12] OF CHAR`

En el caso del ejemplo, una variable de tipo "nombre" debe contener 12 caracteres exactamente. El compilador no añade espacios en blanco para completar una cadena más corta, ni trunca una cadena más larga. Si se especifica una cadena que no tenga esa longitud, se produce un error.

Hay dos formas de construir constructores para estas cadenas:

`'MARIA b PILAR b'`    b = espacio en blanco

`('M', 'A', 'R', 'I', 'A', 'b', 'P', 'I', 'L', 'A', 'R', 'b')`

Con el PASCAL para VAX-11 se pueden incluir caracteres no imprimibles en una cadena de caracteres. Para ello, se cierra la cadena con un apóstrofo, se escribe el ordinal del carácter no imprimible entre paréntesis, y se vuelve a abrir la cadena con otro apóstrofo.

Ejemplo: `'Un pitido '(7)' en una cadena de caracteres'`

### 3.3.2. El tipo RECORD

Un registro es un grupo de componentes llamados campos que pueden ser de tipos diferentes y pueden contener uno o varios elementos de datos. En la definición se especifican el nombre y el tipo de cada campo:

```

TYPE T = RECORD
    s1 : T1;
    s2 : T2;
    .....
    sn : Tn
END;
```

Ejemplos:

```

Complejo = RECORD
    re : REAL ;
    im : REAL
END;
```

```

fecha = RECORD
    día : 1..31 ;
    mes : 1..12;
    año : 1..2000
END;
```

```

persona = RECORD
    apellido : PACKED ARRAY [1..32] OF CHAR ;
    nombre : PACKED ARRAY [1..32] OF CHAR ;
    nacimiento : fecha ;
    sexo : ( varon, hembra ) ;
    ecivil : ( soltero, casado, viudo, divorciado )
END;
```

Los campos de un tipo registro pueden ser también de tipo registro, como en el tercer ejemplo.

Los constructores de registro se forman con el nombre del tipo seguido por valores constantes de los tipos correspondientes a cada campo puestos en el mismo orden en que fueron declarados, separados por comas y encerrados entre paréntesis. Si algún campo es de tipo registro, su constructor queda en un paréntesis anidado.



## Registros con variantes

En la práctica, suele resultar conveniente y natural considerar dos tipos de datos como variaciones del mismo tipo.

Por ejemplo, en un inventario que incluye a equipos de música y televisores, puede interesar anotar la potencia en vatios, para los primeros; y el tamaño de la pantalla y la capacidad de color para los televisores. Según la clase de aparato interesará anotar unas u otras características. Pues bien, para representar este caso puede ser adecuada la estructura de registro con variante que se describe a continuación:

```

TYPE
    aparato = (estereo,tv) ;
    nombre = PACKED ARRAY [1..40] OF CHAR ;
    fichas = RECORD
        nref : 1..20000 ;
        proveedor : nombre ;
        cantidad : INTEGER ;
        CASE clase_aparato : aparato OF
            estereo : (potencia : 1..1000) ;
            tv : (pantalla : 1..25 ; color : BOOLEAN)
        END;

```

En estos casos se introduce un tercer componente, llamado discriminante de tipo o campo indicador, para identificar la variante adoptada por una variable. En el ejemplo, el nombre del discriminante es "clase\_aparato" y quedan definidos los campos siguientes:

```

ficha . nref
ficha . proveedor
ficha . cantidad
ficha . clase_aparato
ficha . potencia
ficha . pantalla
ficha . color

```

La forma general de la definición de un tipo de registro variante es:

```

TYPE
    T = RECORD  s1: T1 ; ... ; sn-1 : Tn-1 ;
                CASE sn : Tn  OF
                    c1 : (s1,1 : T1,1 ; ... ; s1,n1 : T1,n1) ;
                    ....
                    cm : (sm,1 : Tm,1 ; ... ; sm,nm : Tm,nm)
                END

```

Los  $s_i$ ,  $s_{i,j}$  son los nombres de selectores de componentes cuyos tipos son  $T_i$ ,  $T_{i,j}$  y  $s_n$  es el componente del campo discriminante cuyo tipo es  $T_n$ . Las constantes  $c_1, \dots, c_m$  designan valores del tipo ordinal  $T_n$ .

Las variantes sólo se pueden definir para los últimos campos del registro, pero los campos variantes pueden estar anidados como en el ejemplo siguiente:

```

TYPE
    tipo_sexo = (varon, hembra) ;
    fechas = RECORD
        día : 1..31;
        mes : 1..12;
        año : 1..2000
    END;
    paciente = RECORD
        nombre : STRING [32] ;
        nacimiento : fechas;
        CASE sexo : (tipo_sexo) OF
            varon : ( ) ;
            hembra : (CASE madre : BOOLEAN OF
                FALSE : ( ) ;
                TRUE : (hijos : INTEGER )) ;
        END;
    END;

```

El constructor de un tipo registro que contiene variantes debe incluir valores para el campo discriminante y para todos los identificadores de campo correspondientes a cada variante. Se debe especificar un valor para el campo discriminante aunque no se haya declarado identificador de discriminante (esto es posible, aunque no recomendable).

Para los ejemplos anteriores "fichas" y "paciente", se pueden proponer los constructores siguientes:

```

fichas (200, nombre('AESA',36 OF ' '), 20, estereo, 40)
fichas (350, nombre('AESA',36 OF ' '), 30, tv, 14, TRUE)
paciente (nombre('SANCHEZ, S.'), fechas(1,12,1942), varon)
paciente (nombre('PEREZ, P.'), fechas(1,2,1932), hembra, FALSE)
paciente (nombre('LOPEZ, L.'), fechas(10,3,1953), hembra, TRUE, 2)

```

y si la variable "ingresado" es de tipo "paciente", se pueden usar los descriptores de campo siguientes:

```

ingresado . nombre
ingresado . nacimiento
ingresado . nacimiento . dia
ingresado . nacimiento . mes
ingresado . nacimiento . año
ingresado . sexo
ingresado . madre
ingresado . hijos

```

Para el siguiente caso :

```

TYPE
    llamadas = RECORD
        comunicante : nombre ;
        hora : REAL ;
        asunto : (negocio, entrevista, ocio) ;
        CASE BOOLEAN OF
            TRUE : (fecha : fechas) ;
            FALSE : ( ) ;
        END
    END

```

Son posibles estos dos constructores:

llamadas ('JUAN LOPEZ', 10.30, entrevista, TRUE, fechas (2,9,1986))

llamadas ('JUAN LOPEZ', 10.30, negocio, FALSE)

Notar que es necesario incluir un valor para el campo discriminante aunque se haya declarado sin identificador.

Suponiendo que la variable "aviso" es de tipo "llamadas", se pueden manejar los descriptores siguientes:

```

aviso . comunicante
aviso . hora
aviso . asunto
aviso . fecha
aviso . fecha . dia
aviso . fecha . mes
aviso . fecha . anio

```

Respecto a la selección de campos para operar con ellos a lo largo de un programa, hay que tener en cuenta que sólo se puede hacer referencias a los campos de la variante que está activada. Cuando se declara identificador para el campo discriminante, la variante en curso, o variante activada, es aquella que corresponde al caso que representa el valor del discriminante en ese momento. Hasta que se asigne otro valor al identificador de discriminante, no se puede hacer referencia a campos correspondientes a otros casos. Cuando no se declara identificador de discriminante, una referencia a un identificador de campo hace que la variante correspondiente se active, quedando indefinidas las demás inmediatamente.

Por ello, la mejor forma de utilizar los registros variantes consiste en agrupar las operaciones correspondientes a cada variante en una instrucción selectiva, llamada sentencia CASE, cuya estructura es una imagen de la definición del tipo de registro variante:

```

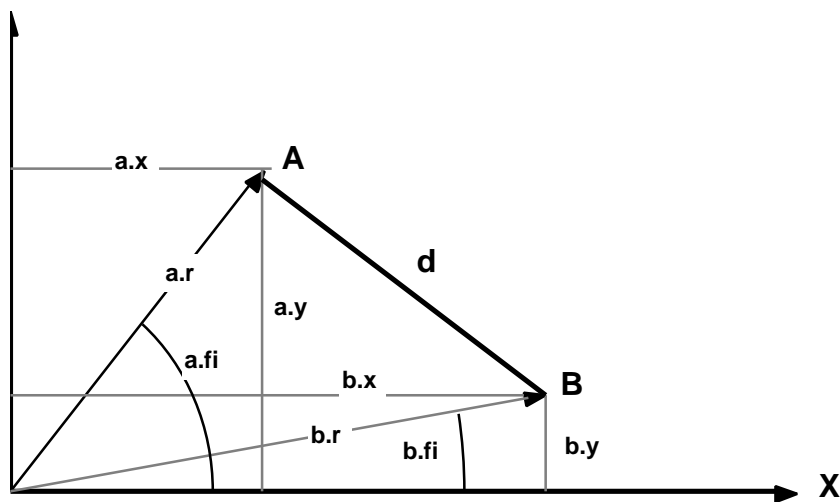
CASE      x.sn  OF
          c1 : S1 ;
          c2 : S2 ;
          ...
          cm : Sm
END

```

$S_k$  representa las instrucciones correspondientes al caso de que  $x$  adopte la forma de la variante  $k$ , es decir, se selecciona su ejecución sólo cuando el campo discriminante  $x.s_n$  toma el valor  $c_k$ .

La sentencia CASE se tratará en otro capítulo, más adelante, pero aquí se va a adelantar un ejemplo para ilustrar sobre el uso de los registros con variantes. Es una parte del programa que tiene por objeto calcular la distancia entre los puntos A y B dados por las variables "a" y "b" del tipo coordenadas:

La parte ejecutable tiene programado el cálculo para todas las combinaciones posibles. No se muestra la sección de lectura de datos ni la de escritura de resultados.



**Figura 3.1**  
**Coordenadas polares y cartesianas**

```

TYPE      coordenadas = RECORD      CASE clase : (cartesiana, polar) OF
                                         cartesiana : (x,y :REAL) ;
                                         polar : (r : REAL ; fi : REAL)

                                         END ;
VAR      a,b : coordenadas;
...
CASE     a.clase OF
  cartesiana : CASE b.clase OF
    cartesiana : d :=SQRT (SQR (a.x-b.x)+SQR (a.y - b.y)) ;
    polar :      d := SQRT (SQR (a.x-b.r *COS (b.fi))
                           + (SQR (a.r * SIN (b.fi)))
                           )
    END ;
  polar :      CASE b.clase OF
    cartesiana : d := SQRT (SQR (a.r * SIN (a.fi) - b.y))
                  + SQR (a.r * COS (a.fi) - b.x)) ;
    polar :      d := SQRT ( SQR (a.r) + SQR (b.r)
                           - 2 * a.r * b.r * COS (a.fi - b.fi))
    END ;
END;
END;

```

### 3.2.3. CADENAS DE LONGITUD VARIABLE.

El PASCAL para VAX-11 aporta EL tipo VARYING OF CHAR que permite manejar cadenas de caracteres de longitud variable desde cero hasta un valor máximo que se indica en la declaración. En otras implementaciones se denomina STRING.

Sintaxis :    VARYING [longitud\_maxima] OF CHAR

              STRING [longitud\_maxima]

Ejemplo :    TYPE nombre = VARYING [36] OF CHAR

              TYPE linea = STRING [80]

Para las variables o componentes de este tipo, el compilador reserva en memoria un espacio suficiente para poder almacenar una cadena de la longitud máxima. Las longitudes de las cadenas asignadas pueden variar desde cero hasta el valor máximo especificado. Una cadena VARYING de longitud cero es la cadena vacía (").

El tipo VARYING OF CHAR tiene algunas propiedades de los tipos ARRAY y RECORD. En realidad, una cadena de longitud variable se almacena igual que si fuese un registro con dos campos : LENGTH (longitud) y BODY (cuerpo).

La sintaxis expuesta antes se puede imaginar como el tipo registro :

RECORD

    length : 0..longitud\_máxima ;

    body : PACKED ARRAY [1..longitud\_máxima] OF CHAR ;

END

Los identificadores de campo LENGTH y BODY están predeclarados en el PASCAL para VAX-11. El campo LENGTH contiene la longitud de la cadena actual, y el campo BODY contiene a la cadena misma. Si se desea, se puede acceder a esos valores como se haría con los valores de los campos del registro.

Para el tipo STRING no hay una sintaxis de constructor característica, se usa la misma que para las cadenas de longitud fija, con la diferencia de que aquí el constructor puede ser más corto que la longitud declarada como máxima.

Ejemplos :

Para el tipo STRING [30] , los siguientes son valores posibles:

    'Fernando Fernandez Fernandez'

    'Pepe'

Para el tipo ARRAY [1..5] OF STRING [20] , un constructor posible:  
( 'Leon', 'Zamora', 'Salamanca', 'Valladolid', 'Palencia' )

Para el tipo:

```
RECORD
    Titulo : STRING [80] ;
    Autor  : STRING [36] ;
    Categoria : (Ficcion, Biografia, Viajes)
END ;
```

un constructor posible :

```
('ACALI', 'Santiago Genoves', Viajes)
```

Como selector de componente, se usa la misma sintaxis que en la estructura "array".

### 3.3.4. El tipo SET.

La tercera estructura fundamental es la estructura conjunto (SET). Se define así :

```
TYPE T = SET OF TO
```

Los valores posibles de una variable x del tipo T son conjuntos de elementos de T<sub>O</sub>. El conjunto de todos los subconjuntos formados por elementos de un conjunto T<sub>O</sub> se denomina conjunto potencia o conjunto de las partes de T<sub>O</sub>. Por tanto el tipo T es el conjunto potencia de su tipo base T<sub>O</sub>.

El tipo base debe ser un tipo ordinal.

Ejemplos :

```
TYPE uno = SET OF 0..128
TYPE dos = SET OF CHAR
TYPE tres = SET OF (desconectada, manual, sin_papel,
                   atascada, desalineada)
```

Un conjunto cuyo tipo base sea entero, no puede tener más de 256 elementos. El ordinal de cada elemento debe ser un valor comprendido entre 0 y 255.

Un constructor de SET se forma encerrando entre corchetes valores constantes tomados de la lista de elementos del conjunto. Los valores que ocupan posiciones consecutivas en la definición del tipo se pueden indicar usando el símbolo de subcampo.

Ejemplos:

```
[0,1,2..9]
['A', 'E', 'I', 'O', 'U']
[ desconectada..desalineada]
```

El conjunto vacío se representa así : [ ]

Dadas las variables, "números", "caracteres", "impresora" pertenecientes a los tipos de este ejemplo:

VAR

números : uno ;  
 caracteres : dos ;  
 impresora : ARRAY [1..6] OF tres

las siguientes son expresiones de asignación de valores válidas:

numeros := [2,4,6,8]  
 caracteres := ['+', '-', '\*', '/']  
 impresora [1] := [desconectada]  
 impresora [3] := [ ]  
 impresora [5] := [manual, sin\_papel, atascada]

Los operadores elementales para las estructuras de tipo SET son los siguientes :

\* intersección de conjuntos  
 + unión de conjuntos  
 - diferencia de conjuntos  
 IN pertenencia a un conjunto

El orden de prioridad coincide con el mostrado siendo decreciente desde arriba hacia abajo.

Al operador IN se le considera operador de relación.

### 3.3.5. La estructura SECUENCIA. El tipo FILE

Las estructuras de datos presentadas hasta aquí ("array", registro, conjunto) tienen una característica común : su cardinalidad es finita siempre que los tipos de sus componentes tengan cardinal finito.

La mayoría de las llamadas estructuras avanzadas (secuencias, árboles, grafos) se caracterizan porque su cardinalidad es infinita. Como veremos, ello tiene consecuencias prácticas importantes.

La estructura SECUENCIA se define así : Una secuencia con tipo base  $T_0$  es, o la secuencia vacía, o la concatenación de una secuencia (de tipo base  $T_0$ ) con un valor de tipo  $T_0$ .

La secuencia, tipo T, definida así, comprende infinitos valores. Cada valor en sí mismo contiene un número finito de componentes de tipo  $T_0$ , pero este número no está acotado porque para cualquier secuencia dada es posible construir otra más larga.

Con otras estructuras avanzadas se pueden hacer consideraciones semejantes. Lo más importante es que la cantidad de memoria necesaria para representar un dato de tipo estructurado avanzado no se conoce en tiempo de compilación : de hecho puede variar durante la ejecución del programa. Esto requiere algún esquema de asignación dinámica en el que la memoria se vaya ocupando conforme los valores crecen y se vaya liberando para otros usos a medida que los valores disminuyen.

La representación de estructuras avanzadas es un problema difícil cuya solución influye en la eficacia de la ejecución de un proceso. Una elección adecuada sólo puede hacerse si se conocen las operaciones elementales que se vayan a realizar sobre la estructura y su frecuencia de ejecución. Como esta información es desconocida para el diseñador del lenguaje y de su compilador, es lógico que las estructuras avanzadas se encuentren excluidas de los lenguajes de uso general.

En la mayor parte de los lenguajes de programación el problema se resuelve teniendo en cuenta que todas las estructuras avanzadas de datos se componen de elementos o de estructuras fundamentales de datos. Cualquier estructura se puede generar mediante operaciones especificadas por el programador siempre que cuente con posibilidades de asignación dinámica de sus componentes y de encadenamiento y referencia dinámica de los mismos. En PASCAL, esto se consigue con los PUNTEROS, que se tratarán más adelante.

La secuencia es una estructura que se puede considerar avanzada porque su cardinalidad es infinita, pero su uso es tan general y frecuente que resulta obligado incluirla entre las estructuras básicas.

En forma análoga a como se hizo con los "arrays" y conjuntos, un tipo fichero (FILE) se define con la fórmula :

$$\text{TYPE T} = \text{FILE OF T}_0$$

Un fichero es una secuencia de componentes del mismo tipo.

El tipo de los componentes puede ser ordinal, real, puntero o un tipo estructurado excepto un tipo FILE o un tipo estructurado con un componente de tipo FILE.

Ejemplos:

```
TYPE texto = FILE OF CHAR
```

```
TYPE archivo =      FILE OF RECORD
                    Titulo : cadena_1 ;
                    Autor  : cadena_2 ;
                    Editorial : cadena_3 ;
                    Fecha  : 1..2000
                    END
```

Los operadores aritméticos, relacionales, lógicos y de asignación no se pueden usar con variables de tipo FILE ni con estructuras que contengan componentes de tipo FILE.

No se pueden formar constructores para los tipos FILE.

Lo más importante del acceso secuencial es que en cada momento sólo se puede acceder de forma inmediata a un único, y determinado, componente de la secuencia. Este componente

corresponde a la posición del mecanismo de acceso en ese momento. Tal posición se puede cambiar, con los operadores de fichero, al componente siguiente o al primero de toda la secuencia.

Otra característica muy importante es que los procesos de construcción y de inspección de una secuencia son distintos y no se pueden mezclar en cualquier orden. Un fichero se construye añadiendo repetidamente componentes a su extremo final y después se puede inspeccionar secuencialmente. Por ello, se considera que un fichero se puede encontrar en estado de ser construido (escritura) o en estado de ser inspeccionado (lectura); pero no puede estar en los dos simultáneamente.

### **Operadores elementales de ficheros secuenciales.**

Aquí se va a concretar la noción de acceso secuencial mediante la definición de un conjunto de operadores elementales utilizables por el programador. Son cuatro operaciones :

1. Iniciar el proceso de generación del fichero.
2. Añadir un componente al final de la secuencia (escritura).
3. Iniciar la inspección (lectura).
4. Avanzar la inspección al componente siguiente.

Las operaciones 2 y 4 (escritura y lectura) se definen con el auxilio de una variable implícita que representa una memoria intermedia de amortiguación llamada "buffer" . Cuando se declara una variable de tipo FILE, el compilador crea automáticamente una variable "buffer" del mismo tipo que el componente.

Para una variable "x" de tipo FILE, el "buffer" se designa por "x^". Si "x" es de tipo T, entonces "x^" es del tipo base T<sub>0</sub>.

En lo que sigue, se considerará a un fichero "x" como si estuviera dividido en dos partes, una "x<sub>I</sub>" a su izquierda y otra "x<sub>D</sub>" a su derecha. Entonces se puede escribir

$$x = x_I \& x_D$$

Con el símbolo & se representa la concatenación.

Con < > se designa la secuencia vacía.

Con <x<sub>0</sub>> se representa la secuencia de un único componente x<sub>0</sub>. Secuencia unitaria.

1. Construcción de la secuencia vacía.  
Se consigue con la operación REWRITE (x), que equivale a la asignación:  
 $x := \langle \rangle$   
Esta operación escribe sobre la secuencia en curso e inicia el proceso de construcción de una nueva secuencia.

## 2. Extensión de una secuencia.

Se consigue con la operación PUT(x), que corresponde a la asignación:

$$x := x \& \langle x^\wedge \rangle$$

que añade el valor del "buffer" ( $x^\wedge$ ) al final de la secuencia x.

## 3. Comienzo de una inspección.

Se consigue con la operación RESET(x), que equivale a las asignaciones simultáneas:

$$x_I := \langle \rangle$$

$$x_D := \langle x \rangle$$

$$x^\wedge := \text{primero}(x) \quad (\text{primer elemento})$$

Este operador se usa para iniciar el proceso de lectura de una secuencia: posicionamiento al comienzo de la secuencia y asignación del primer componente al "buffer".

## 4. Paso al componente siguiente.

Se consigue con la operación GET(x), que equivale a las asignaciones simultáneas:

$$x_I := x_I \& \langle \text{primero}(x_D) \rangle$$

$$x_D := \text{resto}(x_D)$$

$$x^\wedge := \text{primero}(\text{resto}(x_D))$$

Se avanza la posición un lugar y se asigna ese componente al "buffer".

Notar que el valor primero ( $x_D$ ) sólo está definido si el componente es distinto de  $\langle \rangle$ .

Los operadores REWRITE y RESET no dependen de la posición del fichero previa a su ejecución; ambos vuelven a situar el fichero, en cualquier caso, en su posición origen.

## La función EOF

Cuando se inspecciona una secuencia es necesario poder reconocer el final de la misma porque, si no, se alcanzaría el caso de que la asignación

$$x^\wedge := \text{primero}(x_D)$$

representaría una expresión no definida.

Alcanzar el final del fichero es sinónimo de que la parte a la derecha,  $x_D$ , está vacía. Para representar esta condición, el compilador aporta la función predeclarada, EOF ("End of File") de tipo BOOLEAN, que toma el valor "verdadero" (TRUE) en ese caso.

$$\text{EOF}(x) = x_D = \langle \rangle$$

Por tanto, la operación GET(x) sólo se puede ejecutar si EOF(x) es falso.

### Los procedimientos READ y WRITE.

Con los cuatro operadores básicos descritos es posible expresar todas las operaciones sobre ficheros. Pero en la práctica, el compilador ofrece dos procedimientos predeclarados que combinan la operación de avanzar posición en el fichero (GET o PUT) con el acceso a la variable "buffer". Ambos procedimientos se pueden expresar en función de los operadores básicos.

Sean  $v$  una variable y  $e$  una expresión de tipo  $T_0$ , componente del fichero. Entonces se definen:

READ ( $x,v$ ), que es sinónimo de  $v := x^{\wedge}$  ; GET ( $x$ )

o sea :

- . asignar el valor del "buffer" a la variable  $v$
- . avanzar posición en el fichero
- . asignar el nuevo componente al "buffer"

WRITE ( $x,e$ ), que es sinónimo de  $x^{\wedge} := e$  ; PUT ( $x$ )

o sea :

- . asignar el valor de  $e$  al "buffer"
- . añadir el valor del "buffer" al final de la secuencia

Es más cómodo utilizar READ y WRITE en vez de GET y PUT por su brevedad y simplicidad conceptual, ya que de esa forma se puede ignorar la existencia del "buffer".

La variable "buffer" puede ser útil cuando se quiere inspeccionar el componente siguiente sin tener que avanzar la posición en el fichero.

Las condiciones previas necesarias para la ejecución de los dos procedimientos son:

EOF ( $x$ ) = FALSE, para READ ( $x,v$ )

EOF ( $x$ ) = TRUE, para WRITE( $x,e$ )

Al leer, la función EOF toma el valor TRUE (verdadero) tan pronto como se haya leído el último elemento del fichero.

Seguidamente, se indican los esquemas de dos programas para la construcción y lectura secuencial de un fichero  $x$ . Los procedimientos "Leer\_datos", "Procesar\_datos" y la condición controlada por la variable "seguir" se tratarían fuera de estos esquemas.

Para construir el fichero:

```

REWRITE (x) ;
WHILE seguir DO
  BEGIN
    Leer_datos (v) ;
    WRITE (x,v)
  END ;

```

Para leer y procesar el fichero:

```

RESET (x) ;
WHILE NOT EOF (x) DO
  BEGIN
    READ (x,v) ;
    Procesar_datos (v)
  END ;

```

### Los ficheros de texto. El Tipo TEXT

Los ficheros cuyos componentes son caracteres (tipo CHAR) se usan mucho en el proceso de datos porque son la vía de comunicación entre los computadores y los usuarios. Tanto la entrada de datos legible proporcionada por el usuario, como la salida de resultados obtenidos, están formados por secuencia de caracteres.

Es tan importante este tipo de datos, que se le ha dado un nombre normalizado. Es el tipo TEXT que se define así :

```
TYPE TEXT = FILE OF CHAR
```

Es un fichero de caracteres que tiene definidos algunos atributos que se describirán aquí : fin de línea, salto de línea.

En un programa se pueden declarar variables de tipo TEXT a voluntad, pero hay dos que están predeclaradas : INPUT, OUTPUT. Son dos ficheros que están asociados siempre a los dispositivos estándar para la entrada (lectura de datos) y la salida (escritura de resultados) ; INPUT para leer y OUTPUT para escribir. En el trabajo en régimen interactivo, los dos están asociados a la video\_terminal de trabajo. Se declaran en el encabezamiento del programa :

```
PROGRAM p ( INPUT, OUTPUT )
```

Como INPUT y OUTPUT se usan con mucha frecuencia, se ha establecido que sean los ficheros por defecto para la entrada y la salida respectivamente. Es decir, que si no se especifica fichero (primer parámetro) en READ, se usa INPUT ; y si no se especifica en WRITE, se usa OUTPUT.

En forma generalizada, los convenios de notación para los procedimientos READ y WRITE son:

READ ( $x_1, \dots, x_n$ ) es equivalente a READ (INPUT,  $x_1, \dots, x_n$ )  
 WRITE ( $x_1, \dots, x_n$ ) es equivalente a WRITE (OUTPUT,  $x_1, \dots, x_n$ )

y, además :

READ (f,  $x_1, \dots, x_n$ ) es equivalente a  
 BEGIN READ (f,  $x_1$ ) ; ... ; READ (f,  $x_n$ ) END  
 WRITE (f,  $x_1, \dots, x_n$ ) es equivalente a  
 BEGIN WRITE (f,  $x_1$ ) ; ... ; WRITE (f,  $x_n$ ) END

Para conseguir una aproximación cómoda a la noción de texto que manejamos habitualmente, se considera que los ficheros de texto están formados por líneas y se introducen operadores y predicados adicionales para marcarlas y reconocerlas. Se puede pensar que las líneas están separadas por caracteres (hipotéticos) de separación (no pertenecientes al tipo CHAR). En este sentido, el compilador aporta algunos procedimientos :

WRITELN (f)	procedimiento para añadir una marca de línea al final del fichero f.
READLN (f)	procedimiento para saltar caracteres en el fichero f hasta el siguiente inmediato a la próxima marca de línea.
EOLN (f)	función de tipo BOOLEAN que toma el valor TRUE si la posición del fichero ha llegado a una marca de línea, y toma el valor FALSE en otro caso. Se supone que si EOLN (f) es TRUE, entonces f <sup>^</sup> = blanco.

Con esta base se puede proponer un ejemplo para leer texto por INPUT y escribirlo en un fichero f.

```
PROGRAM Copiar (INPUT, OUTPUT) ;
VAR
  f : TEXT ;
  letra : CHAR ;
BEGIN
  REWRITE (f) ;
  WHILE NOT EOF DO
    BEGIN
      WHILE NOT EOLN DO
        BEGIN
          READ (letra) ; WRITE (f,letra)
        END ;
        WRITELN (f) ; READLN
      END
    END
  END.
END.
```

Notar que READLN, WRITELN y EOLN sólo están definidos para las variables de tipo TEXT.

Los compiladores de PASCAL suelen admitir variables de tipo INTEGER o REAL como argumentos de los procedimientos para leer y escribir en ficheros de texto. Esto entra en contradicción con lo anterior. No es coherente porque los valores enteros y reales están representados internamente en binario. Lo que sucede es que los fabricantes incorporan la transformación de las representaciones de datos a los procedimientos para leer y escribir. Si lo tuvieran que hacer los usuarios, se encontrarían siempre con la necesidad de construir procedimientos complejos y lentos para hacer la transformación.

### Ficheros internos y externos.

En cada Sistema Operativo concreto se manejan unidades de almacenamiento de datos en disco llamadas ficheros, que existen fuera del contexto de los programas y se caracterizan mediante atributos propios del Subsistema de Manejo de Ficheros instalado. A éstos se les conoce como ficheros externos.

Las variables de tipo FILE del Pascal son un buen modelo para representar ficheros de disco con acceso secuencial que permite manejarlos aplicando los procedimientos válidos para las secuencias; pero ello obliga a establecer una asociación entre las variables tipo FILE del programa y los ficheros externos identificados con nombres particulares.

Las variables de tipo FILE declaradas sin más, son secuencias que se pueden considerar como ficheros secuenciales sin nombre que no se retienen después de finalizar la ejecución del programa. Son ficheros internos en memoria, que sólo tienen existencia mientras dura la ejecución del programa que los crea.

El ejemplo PAS004 muestra un caso donde se leen datos por la terminal y se almacenan en un fichero interno. Después de haberlo llenado, se recorre leyendo y se muestran sus elementos por la terminal como comprobación. Al terminar la ejecución del programa, el fichero desaparece sin dejar rastro.

```
PROGRAM Fichero_interno (INPUT, OUTPUT) ;
  (* Ejemplo PAS004 *)
  TYPE
    lineas = STRING [80] ;
  VAR
    f : FILE OF lineas ;
    linea : lineas ;
  BEGIN
    REWRITE (f) ;
    WHILE NOT EOF DO
      BEGIN READLN (linea) ; WRITE (f, linea) END ;
    RESET (f) ;
    WHILE NOT EOF (f) DO
      BEGIN READ (f, linea) ; WRITELN (linea) END
    END.
```

En Pascal, la asociación entre las variables de tipo FILE y los ficheros externos se puede conseguir de varias formas que pueden diferir en las implementaciones de los fabricantes. A continuación se describen las técnicas más frecuentes.

### Ficheros especificados en el encabezamiento

Los ficheros externos que se usan en un programa deben estar especificados en el encabezamiento.

Después hay que declararlos como variables de tipo FILE para utilizarlos con los procedimientos típicos RESET, REWRITE, READ, WRITE. Esta técnica es muy común en todas las versiones de Pascal, pero tiene el inconveniente de restringir la validez del programa al caso de los ficheros particulares que se indican en el encabezamiento.

```
PROGRAM Facturacion (INPUT, OUTPUT, Tarifas, Usuario, Utilización);
TYPE
    registro_tarifas = RECORD
        codigo : ARRAY [1..2] OF CHAR;
        precio : REAL
    END;
VAR
    Tarifas : FILE OF registro_tarifas;
    tarifa : registro_tarifas;
...
BEGIN
    ...
    RESET (Tarifas);
    WHILE NOT EOF (Tarifas) DO
        BEGIN
            READ(Tarifas, tarifa);
            Procesar(tarifa);
            ...
        END;
    ...
END.
```

### Asociaciones lógicas

En algunos sistemas operativos es posible establecer asignaciones lógicas fuera del contexto del programa antes del comienzo de la ejecución, para asociar los identificadores indicados en el encabezamiento con otros ficheros externos.

De esta forma se supera la limitación advertida en el caso anterior, puesto que el programa escrito con unos identificadores particulares se puede aplicar a ficheros diferentes cada vez sin tener que volver a escribirlo.

Ejemplo válido para el sistema operativo VMS de VAX-11:

```

PROGRAM Facturacion (INPUT, OUTPUT, Tarifas, Usuarios, Utilizacion);
TYPE
    registro_tarifas = RECORD
        codigo : ARRAY [1..2] OF CHAR;
        precio : REAL
    END;
VAR
    Tarifas : FILE OF registro_tarifas;
...
$ ASSIGN  TARIFENE88.DAT  Tarifas
$ ASSIGN  USERENE88.DAT  Usuarios
$ ASSIGN  ACCENE88.DAT   Utilizacion
$ RUN  Facturacion
...

```

### Asociacion con fichero externo en los procedimientos RESET y REWRITE

En algunas versiones de Pascal, los procedimientos RESET y REWRITE ofrecen la oportunidad para establecer la asociación entre la variable de tipo FILE y un fichero secuencial externo. El formato general se puede expresar así:

```

RESET (v, fichero);
REWRITE (v, fichero);

```

donde "v" representa a la variable de tipo FILE y "fichero" representa a la identificación de un fichero externo, que puede expresarse en forma de constante o de variable.

Si el fichero externo se identifica con una constante, el programa sólo sirve para ese caso.

```

PROGRAM Editor (INPUT, OUTPUT, f);
...
VAR  f : TEXT;
...
RESET (f, 'CAP5'); (* se asocia a "f" con el fichero externo
                    de nombre "CAP5" *)

```

Es más útil emplear una variable para representar al fichero externo porque su valor puede definirse en tiempo de ejecución para que un mismo programa sea aplicable a ficheros diferentes.

```

PROGRAM Editor (INPUT, OUTPUT, f);
...
VAR
    f : TEXT;
    nombre : STRING [32]; (* para almacenar el nombre
                          del fichero *)
...
WRITELN (' Nombre del fichero ? '); READLN (nombre);
RESET (f, nombre); ...

```

### Otros tipos de acceso. La sentencia OPEN

Estamos viendo que con el tipo FILE se pueden manejar ficheros secuenciales (secuencias) de datos almacenados en disco pero, estrictamente, la estructura FILE es una organización de datos en memoria, como todas las demás.

Lo que sucede cuando se establece la asociación entre una variable de tipo FILE y un fichero externo, es que se invoca a las rutinas de librería del Sistema de Manejo de Ficheros para que después, durante la ejecución, se apliquen los procedimientos de lectura y escritura adaptados a los mecanismos de manejo propios de los dispositivos de disco. Esto sucede implícitamente y puede pasar desapercibido al programador que se limita a utilizar las sentencias del lenguaje que emplea.

Para ser riguroso, hay que decir que con los recursos propios del Pascal sólo se puede acceder en modo secuencial a los ficheros de disco. Muchos otros lenguajes ni siquiera ofrecen esta posibilidad.

Para utilizar otros modos de acceso como el modo directo y el modo secuencial indexado, se hace lo mismo que en otros lenguajes. Se invoca al Sistema de Manejo de Ficheros propio del Sistema Operativo en uso, con la sentencia OPEN que está presente en casi todas las versiones del Pascal.

La sentencia OPEN permite abrir un fichero externo en disco, asociandolo a una variable de tipo FILE declarada en el programa, y establecer una serie de atributos relativos al modo de acceso, el estado, el formato, etc. que dependen del Sistema Operativo. Entonces, se dispone de otros procedimientos para leer y escribir que admiten otros parámetros (número de registro para conseguir un acceso directo, clave para fichero indexado, etc.).

Un ejemplo para la versión de Pascal de VAX-11 de D.E.C.:

```
PROGRAM Facturación (INPUT, OUTPUT, Usuarios, Utilización);
...
TYPE      registro_usuarios = RECORD
                                codigo : ARRAY [1..8] OF CHAR;
                                uic   : ARRAY [1..2] OF INTEGER;
                                nombre : ARRAY [1..8] OF CHAR;
                                responsable : ARRAY [1..32] OF CHAR
                                END;
VAR       Usuarios : FILE OF registro_usuarios;
          registro : registro_usuarios;
          nombre   : STRING [32];
...
WRITELN (' Nombre del fichero de usuario ? ');  READLN (nombre);
OPEN (Usuarios, FILE_NAME:=nombre, HISTORY:=OLD,
      ACCESS_METHOD:=KEYED, ORGANIZATION:=INDEXED);
...
RESETK (Usuarios, 0); (* se inicia el modo lectura en el indice primario *)
...
FINDK (Usuarios, 0, '1012PEPE', EQL); (* se localiza un registro *)
READ (Usuarios, registro); (* se lee el registro localizado *)
...
```

Todas estas cuestiones referentes a los modos de acceso a ficheros en disco no se van a tratar con más detalle aquí porque no corresponden al Pascal, sino al entorno operativo de trabajo.

### 3.4. LOS TIPOS PUNTERO

A las variables de los tipos descritos en lo precedente se les llama VARIABLES ESTATICAS. Se declaran y pueden referenciarse posteriormente por su identificador.

Se les llama así porque existen durante la ejecución íntegra del bloque donde están declaradas. Tienen reservado un espacio constante en la memoria permanentemente, mientras existen.

Con los tipos PUNTERO se dispone de la posibilidad de manejar VARIABLES DINAMICAS. Son variables que no se declaran explícitamente, ni pueden ser referenciadas directamente por medio de identificadores; sino que se generan dinámicamente durante la ejecución del programa con el procedimiento NEW . Se les asigna memoria cuando son creadas durante la ejecución del programa, en vez de hacerlo en tiempo de compilación.

Cuando se genera una variable dinámica aparece un valor PUNTERO, que indica la dirección de memoria de la variable que resultó almacenada. Así que un tipo PUNTERO "  $T_p$  " consiste en un conjunto no acotado de valores que apuntan a elementos de un tipo dado "  $T$  ". Entonces se dice que "  $T_p$  " está ligado a "  $T$  ".

La declaración de un tipo PUNTERO tiene esta forma:

```
TYPE  $T_p$  = ^ T
```

y expresa que los valores del tipo "  $T_p$  " son punteros a datos de tipo "  $T$  ".

Para que sea posible la creación de una estructura dinámica, el tipo "  $T$  " debe contener un puntero y tiene que ser RECORD para permitir manejar datos de otros tipos. Los punteros son TIPOS RECURSIVOS.

```
TYPE T = RECORD
    s1 :  $T_p$  ;
    s2 :  $T_2$  ;
    ...
END
```

Si, por ejemplo, "p" es una variable puntero ligada a un tipo " T<sub>p</sub> " por la declaración:

```
VAR p : Tp
```

entonces, "p<sup>^</sup> " denota la variable a la que apunta "p". Para crear valores de "p" se utiliza el procedimiento estandar NEW. La invocación NEW (p) almacena en memoria una variable puntero de tipo " T<sub>p</sub> " y asigna su dirección a "p". Ahora el valor puntero puede ser identificado como "p" . Por otra parte, la variable referenciada por "p" se denomina "p<sup>^</sup>".

Con los tipo puntero se pueden manejar ESTRUCTURAS DINAMICAS que se pueden crear y destruir en cualquier momento de la ejecución del programa y son susceptibles de expandirse y contraerse libremente. En ellas se manejan posiciones de memoria anónimas a las que no se accede por nombre, sino mediante apuntadores o punteros que hacen referencia a ellas. En las ESTRUCTURAS ESTATICAS, se accede a las posiciones de memoria por nombres (nombres de variables) que es necesario declarar y dimensionar en tiempo de compilación. Una estructura estática como la tabla (array) ocupa un número fijo de posiciones de memoria, que se reservan al comienzo de la ejecución ,y no puede cambiar de tamaño.

Las únicas operaciones que están definidas sobre los punteros son la asignación y la comparación por igualdad. Está definido un valor puntero NIL, que apunta al vacío, y pertenece a todos los tipos puntero.

Cuando se ejecuta NEW(p) repetidamente, se generan valores diferentes del puntero "p". Son direcciones de memoria correspondientes a valores diferentes de la variable "p<sup>^</sup>", que es un registro con un campo puntero del mismo tipo que "p".

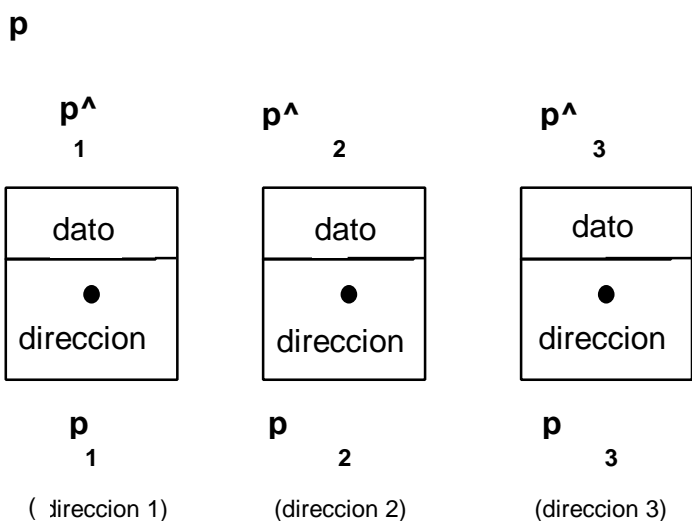
Con estas declaraciones, por ejemplo:

```
TYPE
    puntero = ^ cadena;
    cadena = RECORD
        direccion : puntero;
        dato : CHAR
    END;
VAR
    p : puntero;
```

la situación se puede describir con la representación siguiente, donde se han utilizado subíndices para distinguir cada valor.

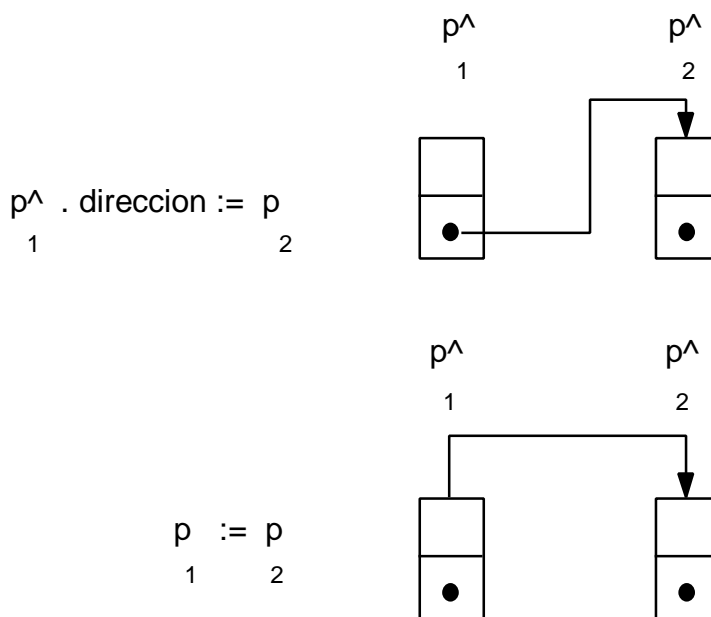
Ahí, "p" es la variable de tipo puntero. Sus valores (p<sub>1</sub>, p<sub>2</sub>, p<sub>3</sub>) apuntan a valores de la variable "p<sup>^</sup>" (p<sup>^</sup><sub>1</sub>, p<sup>^</sup><sub>2</sub>, p<sup>^</sup><sub>3</sub>) que contiene dos campos:

p <sup>^</sup> .dato	, que puede almacenar a un carácter
p <sup>^</sup> .direccion	, que es un puntero con el que se puede apuntar , por ejemplo, a otro elemento de ese conjunto.



La cuestión es que los valores que puede tomar un puntero responden a un nombre genérico. No se pueden distinguir con subíndices o con identificadores diferentes como se acostumbra con otros tipos de variables. Entonces, lo que se hace es ir estableciendo una relación de encadenamiento entre los valores conforme se van generando. Así, se pueden construir estructuras dinámicas de varias clases, que se distinguen por el método a seguir para acceder a sus elementos: COLAS, PILAS, ANILLOS, ARBOLES.

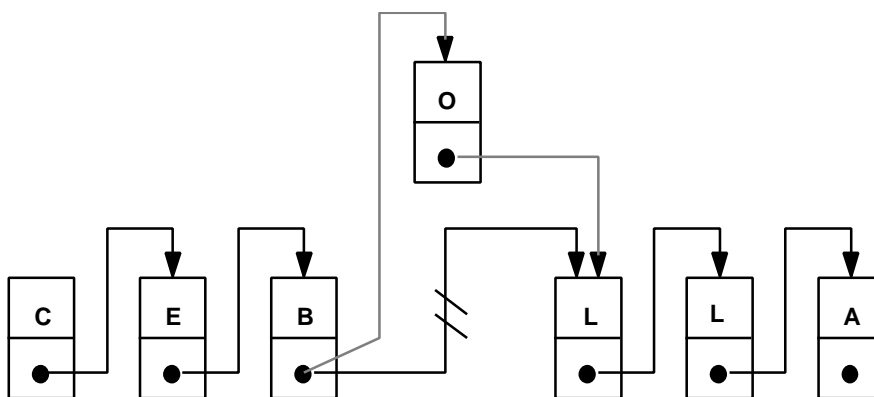
La relación de encadenamiento se establece mediante la sentencia de asignación, que puede adoptar dos formas:



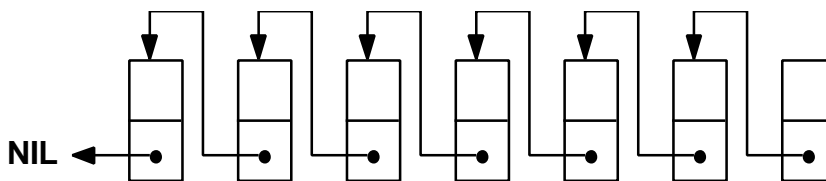
Estas sentencias de asignación se representan con una flecha que indica una conexión y significa el efecto de "estar apuntando a".

Con este modelo, las operaciones de manejo de listas de elementos encadenados, consistentes en añadir, eliminar e insertar elementos, se reducen a ejecutar sentencias de asignación y pueden imaginarse como el establecimiento y ruptura de conexiones entre los elementos implicados.

Para reconocer alguna de las ventajas que aportan los punteros, cabe pensar en el problema de insertar un elemento en una lista. Puede ser el caso de la edición de textos. Un texto se puede considerar como una lista de caracteres (el salto de línea, también es un carácter). Si se utiliza la estructura ARRAY para almacenar a los elementos de la lista, la operación de insertar un elemento resulta costosa porque obliga a desplazar todos los elementos siguientes. Si se almacena la lista en una estructura dinámica donde los elementos están encadenados mediante puntero, la inserción de un elemento se reduce a romper un enlace y construir otros dos enlaces nuevos; lo que resulta mucho más económico en tiempo independientemente del tamaño de la lista y de la posición.



En otro capítulo, se estudian algunas de las operaciones típicas en el manejo de punteros para el mantenimiento de listas. Aquí se presenta sólo un programa completo como ejemplo de manejo de punteros en un caso sencillo. Se trata de crear una lista encadenando elementos que se leen durante la ejecución, y recorrerla después en sentido inverso mostrándolos. El ejemplo se refiere a una lista de caracteres que se leen uno a uno y se organizan en forma de pila (LIFO: el último en entrar es el primero en salir).



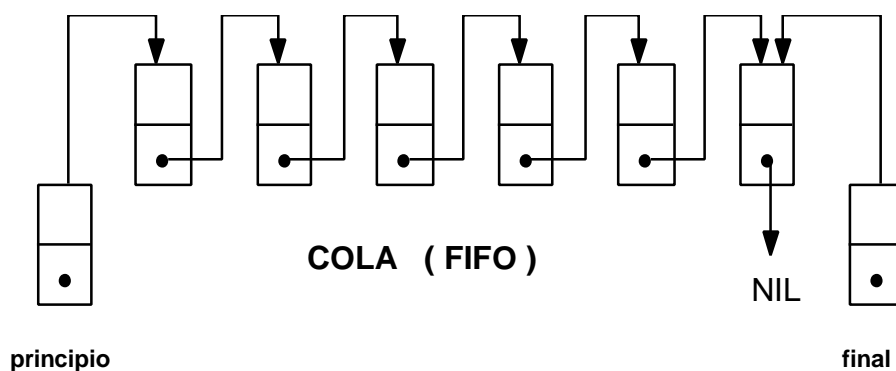
**PILA ( LIFO )**

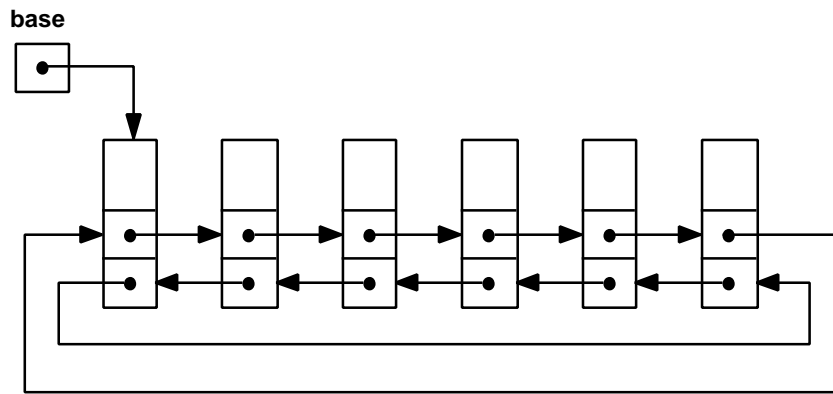
```

PROGRAM Inverlist (INPUT, OUTPUT);
(* Ejemplo PAS009 *)
(**)
TYPE
  puntero = ^ lista;
  lista = RECORD
    direccion : puntero;
    dato : CHAR
  END;
VAR
  base, anterior : puntero;
(**)
(**)
BEGIN
  base := NIL;
  WRITELN ( ' Introduzca la lista: ' );
  WHILE NOT EOLN DO
    BEGIN
      NEW (anterior);
      READ (anterior ^.dato);
      anterior^.direccion := base;
      base := anterior;
    END;
  READLN;
  WHILE anterior <> NIL DO
    BEGIN
      WRITE (anterior ^.dato);
      siguiente := anterior ^.direccion
    END;
  WRITELN;
END.

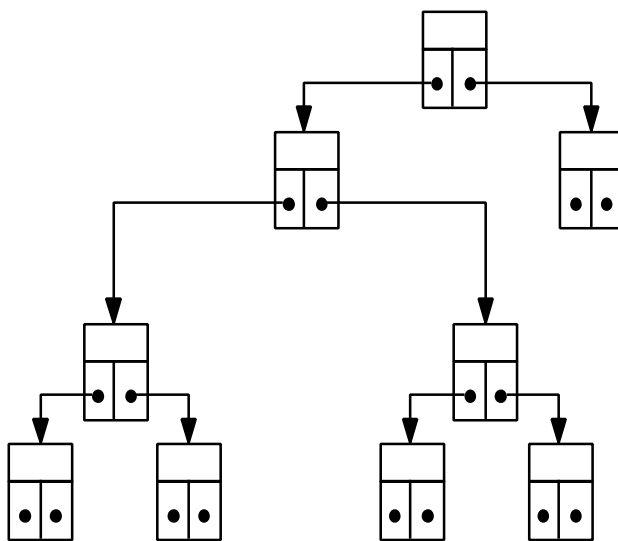
```

A continuación se presentan las representaciones gráficas de algunas otras estructuras de uso común construidas con punteros:





Anillo con doble encadenamiento



ARBOL

## 4. OPERADORES Y EXPRESIONES

Una expresión es una colección de operandos (constantes, variables, funciones) enlazados por ciertos operadores para construir una fórmula algebraica que representa un valor.

Algunas expresiones se evalúan en tiempo de compilación. Por ejemplo: la más simple, que consiste en una constante; o alguna expresión que combine constantes con operadores y funciones predeclaradas.

Otras expresiones se evalúan en tiempo de ejecución. Son las que contienen variables y funciones. Para incluir una función en una expresión basta con escribir su identificador y la lista de parámetros si fueron declarados. En tiempo de cálculo, la función se sustituye por el valor de su resultado.

El PASCAL para VAX-11 admite que las expresiones contengan ciertas combinaciones de operandos de tipos diferentes porque realiza automáticamente la conversión a un mismo tipo antes de hacerse el cálculo.

### 4.1. CONVERSIONES DE TIPO

El PASCAL es un lenguaje muy orientado hacia los tipos de datos y normalmente sólo permite hacer operaciones entre valores del mismo tipo. Pero hay casos donde tiene sentido combinar elementos de tipo diferente porque tienen algo en común. Por ejemplo: sumar un valor de tipo INTEGER con otro de tipo REAL. Esta combinación se admite porque el valor de tipo INTEGER se convierte a su equivalente de tipo REAL antes de realizarse la operación. El resultado será de tipo REAL.

En casos semejantes, la conversión se efectúa automáticamente tanto si se trata de una operación, una asignación o una asociación entre parámetros actuales y formales de las rutinas.

A efectos de conversión, los tipos aritméticos están sometidos a la siguiente ordenación jerárquica :

QUADRUPLE	mayor
DOUBLE	
REAL	
INTEGER	menor

Igualmente lo están los tipos de caracteres:

STRING	mayor
PACKED ARRAY OF CHAR	
CHAR	menor

Cuando se combinan dos valores de tipos diferentes, el operando de menor rango se convierte a su valor equivalente en el tipo del operando de rango mayor. El resultado de una operación donde se produce conversión de tipo es siempre del tipo de mayor rango.

Al asignar una expresión de tipo STRING a una variable de tipo PACKED ARRAY OF CHAR se produce un caso de conversión especial. Si la cadena de tipo STRING tiene exactamente el mismo número de componentes que el ARRAY empaquetado, se convierte la cadena STRING a PACKED ARRAY y después se hace la asignación. En caso contrario, se produce un error.

## 4.2. OPERADORES

El PASCAL proporciona varias clases de operadores para formar expresiones combinando constantes, variables y funciones:

ARITMETICOS  
RELACIONALES  
LOGICOS  
DE CADENAS DE CARACTERES  
DE CONJUNTOS

### 4.2.1. Operadores aritméticos

Un operador aritmético proporciona una fórmula para calcular un valor. La tabla 4.1 los resume.

La suma, resta, multiplicación y exponenciación pueden actuar sobre operandos enteros y reales y producen un resultado del mismo tipo. Si la expresión combina tipos diferentes, se aplican las reglas de conversión.

En muchas versiones también está definida la exponenciación que se suele representar con el símbolo "\*\*".

Operador	Ejemplo	Resultado
+	A+B	Suma de A y B
-	A - B	B restado de A
*	A * B	Producto de A y B
/	A / B	A dividido por B
DIV	A DIV B	Cociente entero de dividir A
REM	A REM B	Resto de dividir A por B
MOD	A MOD B	Módulo de A con respecto a B

**Tabla 4.1**  
**Los operadores aritméticos**

Cuando se usa un entero negativo como exponente, la exponenciación puede producir resultados inesperados. En la tabla 4.2 se definen los resultados para los casos de base entera.

Base	Exponente	Resultado
0	Negativo o 0	Error
1	Negativo	1
-1	Negativo e impar	-1
-1	Negativo y par	1
Cualquier otro entero	Negativo	0

**Tabla 4.2**  
**Resultados con base entera y exponente negativo**

Por ejemplo:

$$1 ** (-3) = 1$$

$$(-1) ** (-3) = -1$$

$$(-1) ** (-4) = 1$$

$$3 ** (-3) = 0$$

El operador de la división se puede aplicar a enteros y reales, pero siempre produce un resultado real. Por tanto, se puede perder precisión cuando los operadores son enteros.

Los operadores DIV, REM y MOD sólo se pueden aplicar a enteros.

**DIV** divide dos enteros y produce un resultado entero: el cociente entero truncando la parte fraccionaria.

Por ejemplo:  $23 \text{ DIV } 12 = 1$   
 $(-5) \text{ DIV } 3 = -1$

**REM** devuelve el resto de la división entre dos enteros

Ejemplos:  $5 \text{ REM } 2 = 1$   
 $3 \text{ REM } 3 = 0$   
 $(-4) \text{ REM } 3 = -1$

**MOD** devuelve el módulo del primer operando respecto al segundo. El resultado de la operación  $A \text{ MOD } B$  sólo está definido cuando B es un entero positivo. El resultado es siempre un entero comprendido entre 0 y B-1.

El módulo de A con respecto a B se calcula como sigue:

- Si A es mayor que B, B se resta de A respectivamente hasta que el resultado sea un entero positivo menor que B.
- Si A es menor que cero, B se suma a A repetidamente hasta que el resultado sea un entero positivo menor que B.
- Si A es menor que B o igual a cero, entonces el resultado es A.

Ejemplos:  $5 \text{ MOD } 3 = 2$   
 $(-4) \text{ MOD } 3 = 2$   
 $2 \text{ MOD } 5 = 2$

Cuando los dos operandos son positivos, los operadores REM y MOD devuelven el mismo valor.

Los operadores negativos deben encerrarse entre paréntesis para asegurarse de que se interpretan correctamente en la operación MOD y en la exponenciación. Por ejemplo:

$(-2.0) ** 2 = 4.0$   
 $-2.0 ** 2 = -4.0$

En el segundo caso, el signo menos afecta a toda la expresión

### 4.2.2. Operadores relacionales

Un operador relacional compara dos expresiones de tipos ordinal, real, cadena de caracteres o conjunto; y produce un resultado de tipo lógico: TRUE (verdadero), si se cumple la relación; o FALSE (falso), si no se cumple.

La tabla 4.3 resume los operadores relacionales que se pueden aplicar a operadores aritméticos.

Notar que los símbolos dobles que representan a un operador no pueden separarse por espacios en blanco.

Operador	Ejemplo	Resultado
=	A = B	TRUE si A es igual a B
<>	A <> B	TRUE si A es distinto de B
<	A < B	TRUE si A es menor que B
<=	A <= B	TRUE si A es menor o igual
>	A > B	TRUE si A es mayor que B
>=	A >= B	TRUE si A es mayor o igual

**Tabla 4.3**  
**Los operadores relacionales**

### 4.2.3. Operadores lógicos

Un operador lógico evalúa una o más expresiones lógicas (BOOLEAN) y produce otro valor lógico. En la tabla 4.4 se indican todos.

Operador	Ejemplo	Resultado
AND	A AND B	TRUE si A y B son TRUE
OR	A OR B	TRUE si uno de los dos, A o B
NOT	NOT A	TRUE si A es FALSE

**Tabla 4.4**  
**Los operadores lógicos**

Notar que los operandos AND y OR combinan dos condiciones formando una condición compuesta. El operador NOT invierte el valor de una condición simple.

#### 4.2.4. Operadores de cadenas de caracteres

Permiten unir o comparar expresiones de cadenas de caracteres. Su resultado es una cadena o un valor lógico. En la tabla 4.5 se muestran todos:

Operador	Ejemplo	Resultado
+	A+B	Cadena que es la concatenación de las
=	A=B	TRUE si las cadenas A y B
< >	A< >B	TRUE si las cadenas A y B tienen
<	A<B	TRUE si el valor ASCII de la cadena A
< =	A<=B	TRUE si el valor ASCII de la cadena A
>	A>B	TRUE si el valor ASCII de A es mayor
> =	A>=B	TRUE si el valor ASCII de A es mayor

**Tabla 4.5**  
**Operadores con cadenas de caracteres**

Con el signo (+) se pueden unir cualquier combinación de cadenas de longitud variable (STRING), "arrays" empaquetados y caracteres simples.

Los operadores relacionales sólo se pueden aplicar a cadenas de la misma longitud. En las cadenas de longitud variable (STRING) es la longitud actual, no la máxima, la que se considera.

Al comparar dos cadenas, se busca el primer elemento diferente. El resultado depende de los ordinales de los caracteres correspondientes. Por ejemplo, el resultado de la expresión:  
'FERNANDO ' > 'FERNANDEZ'

es verdadero (TRUE).

#### 4.2.5 Operadores de conjuntos

Los operadores de conjuntos permiten obtener la unión, la intersección o la diferencia entre dos conjuntos; comparar dos conjuntos y chequear la pertenencia de un valor ordinal a un conjunto. El resultado es un conjunto o un valor lógico (BOOLEAN).

En la tabla 4.6 se indican todos.

Operador	Ejemplo	Resultado
+	A+B	Conjunto unión de A y B
*	A * B	Conjunto intersección de A y B
-	A-B	Conjunto de los elementos de A que no
=	A=B	TRUE si el conjunto A es igual al
<>	A<>B	TRUE si el conjunto A no es igual al
<=	A<=B	TRUE si A es un subconjunto de B
>=	A>=B	TRUE si B es subconjunto de A
IN	C IN B	TRUE si C es elemento del conjunto B

**Tabla 4.6**  
**Los operadores de conjuntos**

### 4.3. LA PRIORIDAD DE LOS OPERADORES

Los operadores que contiene una expresión establecen el orden en el que se combinarán los operandos, ya que se ejecutan según un orden de prioridad que se muestra en la tabla 4.7

Operador	Prioridad
NOT	Mayor
**	
*, /, DIV, REM, MOD, AND	
+, -, OR	
=, <>, <, <=, >, >=, IN	Menor

**Tabla 4.7**  
**Orden de prioridad de los operadores**

Los operadores con igual prioridad se ejecutan de izquierda a derecha.

Con las expresiones que contienen operadores relacionales, deben usarse los paréntesis para que se evalúen correctamente. Por ejemplo, la expresión :

$$A <= X \text{ AND } B <= Y$$

sería interpretada como  $A <= (X \text{ AND } B) <= Y$  y produciría error si X y B no son de tipo BOOLEAN. Para que se calcule bien, hay que escribirla así:

$$(A <= X) \text{ AND } (B <= Y)$$

Los paréntesis se pueden usar siempre para forzar un orden particular en la evaluación de las expresiones.

En PASCAL, algunas operaciones lógicas son evaluadas sólo parcialmente si el resultado puede determinarse sin necesidad de completar el cálculo. En estos casos, el orden seguido en la evaluación no afecta a la consecución del resultado correcto. Sin embargo, debería tenerse en cuenta siempre la importancia que tiene el orden seguido en la evaluación de las subexpresiones cuando se escriban operaciones lógicas que afecten a funciones con efectos laterales (un efecto lateral puede ser la asignación a una variable global o a un parámetro VAR dentro del bloque de la función).

Como ejemplo, se puede considerar la sentencia siguiente:

```
IF F(A) AND F(B)
THEN ...
```

en este caso, independientemente de cual sea la función que se evalúa primero, si el resultado es FALSE, no se necesita evaluar la otra función porque el resultado del test será FALSE necesariamente. Supongamos que la función F asigna el valor de su parámetro a una variable global; entonces resulta que como no se sabe cual es la función que se ha evaluado antes, no se puede asegurar cual será el valor de la variable global después de ejecutarse la sentencia IF.

## 5. LA SECCION DE LAS DECLARACIONES.

Aunque ya se trató ligeramente este tema en el capítulo 2, aquí se va a retomar para hacer algunas puntualizaciones adicionales.

Todas las entidades establecidas por el usuario deben estar definidas en el bloque de las declaraciones, que se escribe a continuación del encabezamiento del programa, y puede contener varias secciones correspondientes a las etiquetas, constantes, tipos, variables, procedimientos y funciones. Cada sección se abre con una de las palabras reservadas siguientes:

LABEL  
CONST  
TYPE  
VAR  
PROCEDURE  
FUNCTION

No es necesario que todas las secciones estén presentes en un programa. Cuando haya varias, pueden aparecer en cualquier orden pero es preferible el de la enumeración anterior.

En este capítulo se tratarán las secciones LABEL, CONST, TYPE y VAR. Las declaraciones de procedimientos y funciones se tratarán en el capítulo 7, dedicado por entero a las rutinas.

### 5.1 LA DECLARACION DE ETIQUETAS

Las etiquetas son enteros decimales comprendidos entre 0 y MAXINIT que se utilizan para marcar sentencias y hacerlas accesibles con la sentencia GOTO de bifurcación incondicional.

Una etiqueta se define escribiéndola delante de alguna sentencia con el símbolo (:) como separador:

```
...  
GOTO 99 ;  
...  
99 : WRITELN ('FIN') ;  
...
```

Sintaxis :                LABEL etiqueta, ... ;

Ejemplo :                LABEL 10, 100, 999 ;

La declaración y la definición de una etiqueta debe hacerse al mismo nivel en el programa. Cada etiqueta debe ir precediendo exactamente a una sentencia dentro del alcance de su declaración.

No es frecuente utilizar etiquetas porque la sentencia GOTO se suele descartar ya que rompe la estructura de los programas. En algunas implementaciones del PASCAL no existe.

## 5.2 LA DECLARACION DE LAS CONSTANTES.

En la sección CONST se pueden definir constantes simbólicas asociando identificadores de constante con expresiones evaluables en tiempo de compilación.

Sintaxis :                CONST identificador = expresión ; ...

Ejemplos:                CONST anio = 1986 ;  
                                      mes = 'agosto' ;  
                                      pi = 3.1416 ;  
                                      cero = (0, 0, 0) ;

Un identificador de constante mantiene el valor asociado durante la ejecución del programa. Sólo se puede cambiar modificando la definición en la sección CONST.

No se puede acceder a los componentes o campos de una constante que representa un constructor de "array" o de registro.

La utilización de identificadores de constantes es muy recomendable porque facilita la lectura, comprensión y modificación de los programas. Si hay que cambiar el valor de una constante simbólica, basta con modificar su declaración en la sección CONST sin necesidad de cambiar las sentencias que la contienen. Así resulta más fácil el mantenimiento de los programas y su transporte a otras máquinas.

## 5.3. LA DECLARACION DE LOS TIPOS.

Todos los tipos definidos por el usuario deben ser declarados estableciendo sus identificadores y el conjunto de valores permitidos.

Sintaxis:                TYPE identificador = tipo ; ...

Ejemplos:

TYPE

```
vocales = ('A', 'E', 'I', 'O', 'U') ;
dias_del_mes = 1..31 ;
cadena = PACKED ARRAY [1..20] OF CHAR ;
nombre_completo = RECORD
    nombre : cadena ;
    apellido_1 : cadena ;
    apellido_2 : cadena
END ;
ficha = RECORD
    nombre : nombre_completo ;
    profesion : cadena ;
    telefono : INTEGER
END ;
archivo = FILE OF ficha ;
mes = SET OF dias_del_mes ;
```

Un identificador de tipo debe estar definido para poder ser usado en otras deficiones de tipo. La única excepción a esta regla permite usar un identificador de tipo base en una definición de tipo puntero, antes de definir el tipo base; pero el tipo base debe estar definido en la misma sección de declaración de tipos donde se mencionó por primera vez.

Por ejemplo:

TYPE

```
puntero = ^pelicula ;
nombre = PACKED ARRAY [1..20] OF CHAR ;
pelicula = RECORD
    titulo, director : nombre ;
    anio : INTEGER ;
    estrellas : FILE OF nombre ;
    siguiente : puntero ;
END ;
```

#### 5.4. LA DECLARACION DE LAS VARIABLES.

En la sección VAR se declaran todas las variables asociando a cada una un identificador, un tipo y, opcionalmente, un valor inicial.

Sintaxis :                   VAR   identificador, ... : tipo := valor ; ...

Se pueden combinar varios identificadores en una misma declaración de variable si se trata de variables del mismo tipo. Si se declara valor inicial, afectará a todas.

La inicialización de las variables está regida por las reglas siguientes :

1. Sólo se pueden inicializar las variables de localización estática. Las que se declaran a nivel del programa son estáticas por defecto. Para poder inicializar una variable en la declaración a un nivel más interno, hay que asignarle el atributo `STATIC` (extensión de `VAX-PASCAL`).
2. Para inicializar una variable hay que utilizar una expresión evaluable en tiempo de compilación de un tipo compatible. Las variables escalares requieren constantes escalares, las variables estructuradas requieren constructores constantes.
3. No se pueden inicializar las variables de tipo `FILE`.
4. El identificador `NIL` es la única constante que se puede usar para inicializar variables de tipo puntero.

Hacer referencia a una variable en un programa, consiste en usarla en alguna de las situaciones siguientes :

- . La variable o uno de sus componentes se escribe a la izquierda de una sentencia de asignación (`:=`). La referencia se mantiene durante la ejecución de la sentencia.
- . La variable o uno de sus componentes se pasa a una rutina como parámetro `VAR`. La referencia se mantiene durante la llamada a la rutina.
- . La variable o uno de sus componentes es accedida por una sentencia `WITH`. La referencia se mantiene durante la ejecución de la sentencia.

Ejemplos de declaraciones :

`VAR`

```
i, j, k : INTEGER := 0 ;  
error, seguir, terminar : BOOLEAN ;  
archivo : FILE OF PACKED ARRAY [0..80] OF CHAR ;  
radio, superficie : REAL ;  
fecha : fechas := (4, 9, 1986) ;
```

## 6. LAS SENTENCIAS EJECUTABLES.

La parte ejecutable de un programa está formada por sentencias que constituyen la expresión de algoritmos que, aplicados a las estructuras de datos diseñadas, producen acciones enfocadas a la obtención de resultados útiles.

En PASCAL, las sentencias ejecutables se pueden clasificar en SIMPLES y ESTRUCTURADAS.

- |                            |   |
|----------------------------|---|
| Sentencias simples :       | - asignación.<br>- sentencia vacía.<br>- llamada a procedimiento.<br>- sentencia GOTO.  |
| Sentencias estructuradas : | - sentencia compuesta.<br>- condicionales : IF-THEN-ELSE<br>CASE.<br>- repetitivas : WHILE<br>REPEAT<br>FOR.<br>- sentencia WITH. |

Las sentencias estructuradas contienen a sentencias simples o a otras sentencias estructuradas que deben ser ejecutadas en orden, de forma repetitiva o cuando se cumplen condiciones determinadas.

Una sentencia estructurada se puede usar en cualquier lugar donde está permitida una sentencia simple. En lo sucesivo se usará el término sentencia para designar a una sentencia simple o estructurada indistintamente.

## 6.1. LA SENTENCIA DE ASIGNACION.

Reemplaza el valor actual de una variable o función por otro valor especificado por una expresión.

Sintaxis :    identificador := expresión

En tiempo de ejecución, se evalúa la expresión y el resultado se asigna a la variable representada a la izquierda del signo ( := ).

Como identificador se puede usar el nombre de una función o de cualquier variable, excepto de tipo FILE.

La variable (o función) y la expresión deben ser del mismo tipo, admitiéndose sólo las siguientes excepciones:

1. que el tipo de la variable sea real y el tipo de la expresión: entero o un subrango de entero
2. que el tipo de la expresión sea un subrango del tipo de la variable y viceversa

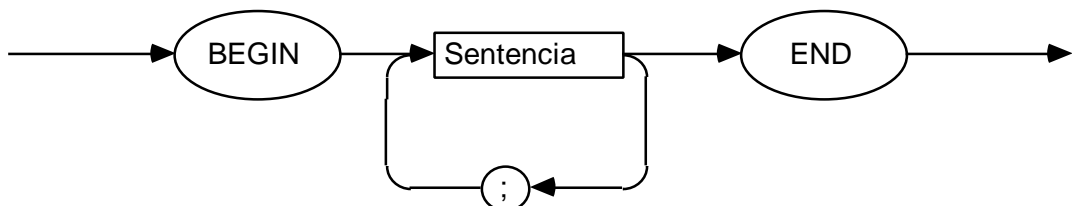
Notar que en el PASCAL se distingue entre el operador de asignación ( := ) y el signo de igualdad ( = ).

Ejemplos :                    N := 1 ;  
                                   T := A < B ;  
                                   vocales := ['A', 'E', 'I', 'O', 'U'] ;  
                                   vector\_1 [1] := vector\_1 [7] + vector\_2 [14] ;  
                                   cliente.nombre := datos.nombre ;

## 6.2. LA SENTENCIA COMPUESTA.

Es una agrupación de sentencias que pueden ejecutarse secuencialmente como si fuesen una sentencia simple.

Sintaxis :                    BEGIN  
                                   sentencia ; ....  
                                   END



Se pueden agrupar sentencias simples y estructuradas, que pueden ser compuestas a su vez.

Las sentencias componentes deben de estar separadas por el delimitador (;). No es necesario escribirlo entre la última sentencia y el delimitador END, pero es recomendable hacerlo para que no se olvide en caso de añadir otras sentencias posteriormente.

### 6.3. LA SENTENCIA VACIA

No produce otro efecto que avanzar el control de la ejecución del programa a la sentencia siguiente.

Se puede representar por dos delimitadores (;) seguidos.

Un caso de sentencia vacía se presenta cuando se escribe el punto y coma después del último componente de una sentencia compuesta. También se suele presentar en las sentencias IF-THEN-ELSE anidadas.

### 6.4. LAS SENTENCIAS CONDICIONALES

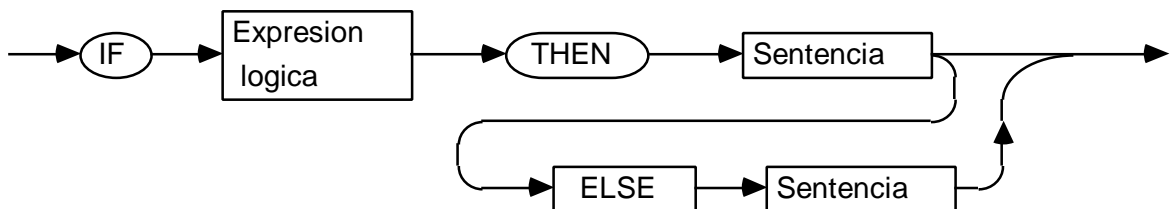
Producen la ejecución de una sentencia según el valor de una expresión de control. Son dos : IF-THEN-ELSE y CASE.

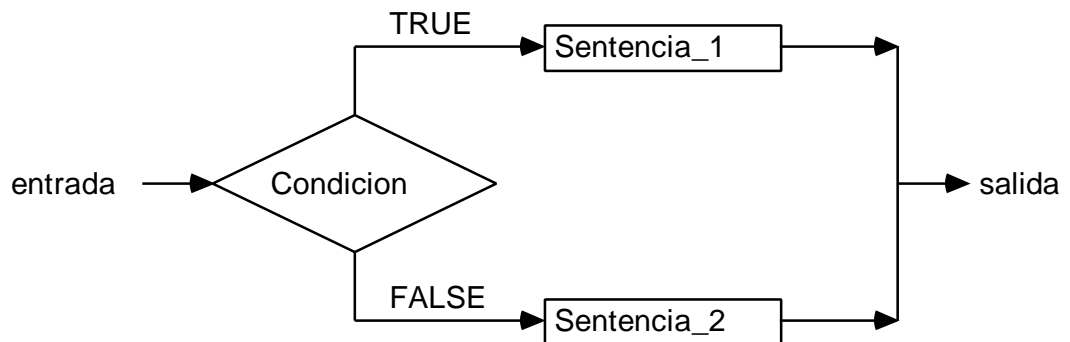
#### 6.4.1 La sentencia IF-THEN-ELSE

Sintaxis:

```

IF  expresión_lógica
THEN
    sentencia_1
ELSE
    sentencia_2
    
```





IF, THEN y ELSE son palabras reservadas.

La expresión que sigue a IF representa la condición a evaluar. La sentencia que sigue a THEN se ejecuta sólo si el valor de la condición es TRUE; en caso contrario se ejecuta la sentencia que sigue a ELSE (o no se ejecuta ninguna si no hay parte ELSE). La sentencia detallada de operaciones es la siguiente:

1. Evaluar la condición.
2. Si es TRUE : ejecutar la parte THEN, y seguir con el paso 4.
3. Si es FALSE : ejecutar la parte ELSE, y seguir con el paso 4.
4. Continuar con el resto del programa. No se puede suponer nada sobre el valor de la condición.

El objeto de una cláusula THEN o ELSE puede ser una sentencia simple o estructurada, incluso otra sentencia IF-THEN-ELSE. Por ejemplo:

```

IF seguir
THEN
  IF B <> 1
  THEN C := 1
  ELSE D := 1 ;
  
```

Este caso se interpreta como si los delimitadores BEGIN y END estuvieran incluidos:

```

IF seguir
THEN
  BEGIN
    IF B <> 1
    THEN
      C := 1
    ELSE
      D := 1
  END ;
  
```

Notar que después de THEN y ELSE no debe escribirse el punto y coma. Si se escribe precediendo a ELSE, se terminará allí la sentencia IF y se producirá un error de compilación.

Cuando se anidan varias sentencias IF, hay que tener en cuenta que la cláusula ELSE afecta a la IF-THEN más próxima.

Ejemplo:

```
IF A=1
THEN
  IF B<> 1
  THEN C := 1
ELSE
  C := 0 ;
```

En este caso, independientemente de la forma de escribir las sentencias, la cláusula ELSE quedará asociada con la sentencia " IF B<> 1 THEN C :=1 ", de forma que si la condición A=1 es falsa, no se tomará ninguna acción. Para que se ejecute la cláusula ELSE cuando A=1 es falso, se puede insertar una sentencia vacía :

```
IF A=1
THEN
  IF B<> 1
  THEN
    C := 1
  ELSE
    ELSE
    C :=0 ;
```

Ejemplo 1 : Notar, en este caso, que la sentencia WRITELN se ejecuta siempre, independientemente de la condición.

```
IF nota >= 5
THEN
  calificacion := 'APROBADO'
ELSE
  calificacion := 'SUSPENSO' ;
WRITELN (calificacion) ;
```

Ejemplo 2: Aquí se ha cometido un error frecuente por no haber previsto el caso de que "numtotal" valga cero. Entonces se originaría una división por cero que produce error.

```
IF (numtotal <> 0) AND (total DIV numtotal > umbral) THEN
  WRITELN ('Promedio aceptable')
ELSE
  WRITELN ('Promedio inaceptable') ;
```

Para resolverlo, se pueden separar las dos partes de la condición :

```
IF numtotal <> 0 THEN
    IF total DIV numtotal > umbral THEN
        WRITELN ('Promedio aceptable')
    ELSE
        WRITELN ('Promedio inaceptable') ;
```

Recordar que la cláusula ELSE va con la sentencia IF-THEN más próxima. Si existen dudas sobre la correspondencia entre las ELSE y las IF, se pueden utilizar los delimitadores BEGIN y END. En tal caso puede quedar:

```
IF numtotal < > 0 THEN
    BEGIN
        IF total DIV numtotal > umbral THEN
            WRITELN ('Promedio aceptable')
        ELSE
            WRITELN ('Promedio inaceptable')
    END ;
```

Ejemplo 3: Aquí se presenta un programa completo para obtener las soluciones de una ecuación de segundo grado.

```
PROGRAM Ecuacion_cuadrática (INPUT, OUTPUT) ;
(* Ejemplo PAS005
Calcula las soluciones de una ecuación de la forma:  $ax^2 + bx + c = 0$  *)
VAR      a, b, c, discriminante, re, im :REAL ;
BEGIN
    READLN (a, b, c)
    IF (a = 0) AND (b = 0)
    THEN WRITELN ('Ecuación degenerada')
    ELSE IF a = 0
        THEN WRITELN ('La solución única es', -c / b)
        ELSE IF c = 0
            THEN WRITELN ('Las soluciones son', -b / a, 'y', 0)
            ELSE BEGIN
                re := -b / (2 * a) ;
                discriminante := SQR (b) - 4 * a * c ;
                im := SQRT (ABS (discriminante)) / (2 * a) ;
                IF discriminante >= 0
                THEN WRITELN ('Soluciones: ',re+im,'y', re-im)
                ELSE BEGIN
                    WRITELN ('Las soluciones son complejas : ');
                    WRITELN (re,'+',im, 'i','y', re,'-',im, 'i')
                END
            END
        END
    END .
```

En el caso de que  $b^2$  sea mucho mayor que  $4ac$ , este programa suele producir resultados falsos.

Si  $b^2$  es mucho mayor que  $4ac$ , entonces el discriminante es aproximadamente igual a  $b^2$  y una de las soluciones será muy pequeña. El cálculo de la solución menor implica la resta de dos números casi iguales, y puede producirse una pérdida de precisión.

En estos casos, sería mejor calcular primero el valor de la solución mayor y después obtener el menor utilizando la relación :

$$\text{solución\_menor} = c / (a * \text{solución\_mayor})$$

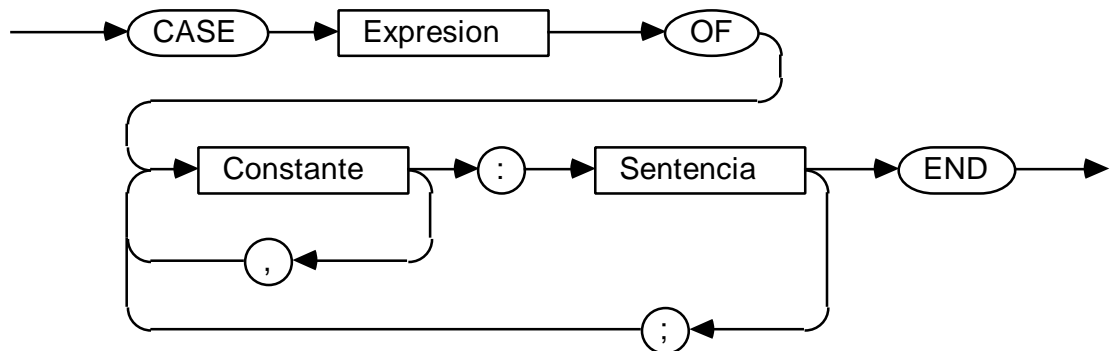
#### 6.4.2. La sentencia CASE

Es una generalización de la sentencia IF-THEN-ELSE.

La sentencia CASE permite elegir la ejecución de una entre varias posibles sentencias según el valor de una expresión ordinal llamada selector.

En ocasiones facilita una escritura más cómoda que usando nidos de IF-THEN-ELSE.

Sintaxis: CASE selector OF  
           lista\_de\_casos : sentencia ; ...  
           END



En el PASCAL para VAX-11 se admite la cláusula OTHERWISE, que es opcional

```

CASE selector OF
  lista_de_casos : sentencia ; ...
  ; OTHERWISE
                 sentencia ; ...
END
  
```

En "lista\_de\_casos" puede ir uno o más valores constantes del mismo tipo ordinal que el selector, separados por comas. Cada caso corresponde a una sentencia que será ejecutada si el valor del selector coincide. Los casos pueden escribirse en cualquier orden. Cada uno puede aparecer una sola vez en una sentencia CASE, aunque puede estar en otras.

En tiempo de ejecución, se evalúa la expresión del selector y se elige una sentencia para ejecutar. Si el valor del selector no aparece en la lista, se ejecuta la sentencia de la cláusula OTHERWISE. Si se omite la cláusula OTHERWISE, el valor del selector debe ser igual a alguno de los casos.

Ejemplo 1 :

```

TYPE
  meses = ( ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP,
  años = 1900..2000 ;
  días_mes = 28..31 ;
VAR
  anio : años ;
  mes : meses ;
  días : días_mes ;
...

CASE mes OF
  ENE, MAR, MAY, JUL, AGO, OCT, DIC : días := 31 ;
  ABR, JUN, SEP, NOV : días := 30 ;
  FEB :      IF (anio MOD 4 = 0) AND (anio MOD 100 <> 0)
              THEN días := 29
              ELSE  días := 28
END ;
...

```

Ejemplo 2 :

```

VAR tipo_letra : (vocal, consonante) ; letra : CHAR ;
...
CASE letra OF
  'A', 'E', 'I', 'O', 'U' :      tipo_letra := vocal ;
  OTHERWISE                  tipo_letra := consonante
END ;
...

```

Ejemplo 3 :

```

VAR control : 1..20 ;
...
IF control IN [2, 3, 5, 7, 8, 11, 17, 20]
THEN
  CASE control OF
    2, 5 :    Calcular (1) ;
    3, 7, 11 : Calcular (2) ;
    8, 17 :   Calcular (3) ;
    20 :     Calcular (4)
  END
ELSE WRITELN (' Valor de control ilegal ');
...

```

Ejemplo 4 :

```

VAR
  dia : (lunes, martes, miercoles, jueves, viernes, sabado, domingo) ;
...
CASE dia OF
  domingo : ;
  lunes, martes, miercoles, jueves, viernes :
    BEGIN
      Ir_a_trabajar ;
      Trabajar ;
      Volver_a_casa ;
    END ;
  sabado : lavar_el_coche
END ;
...

```

## 6.5. LAS SENTENCIAS REPETITIVAS

Permiten programar la repetición de un grupo de sentencias mediante la construcción denominada CICLO o BUCLE. El grupo de sentencias que tienen que repetirse se llama "rango" del ciclo. El número de veces que ha de repetirse el rango está determinado por la "sentencia de control" de ciclo. También suele existir una sentencia "final de ciclo" que indica el final de las sentencias que componen el rango.

Hay dos formas de ciclo : el ciclo condicional y el ciclo con contador.

El ciclo condicional puede tener dos formas:

a) Repetir mientras condición

```

sentencia_1
sentencia_2
...
sentencia_n

```

b) Repetir

```

sentencia_1
sentencia_2
...
sentencia_n
    
```

hasta condición

El ciclo con contador tiene la forma :

```

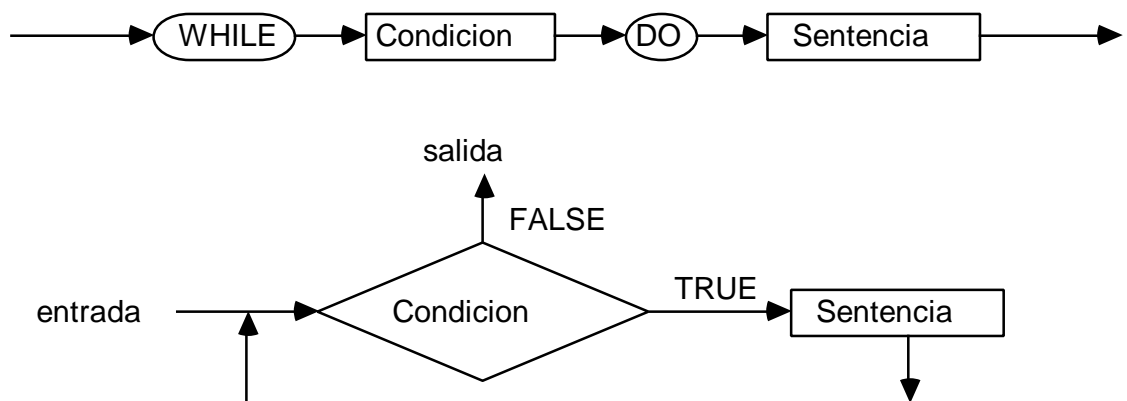
Repetir para nombre = valor_inicial, valor_inicial+incremento ..., valor_final
sentencia_1
sentencia_2
...
sentencia_n
    
```

En PASCAL, el ciclo condicional se consigue con las sentencias WHILE y REPEAT; y para el ciclo con contador está la sentencia FOR.

### 6.5.1. La sentencia WHILE.

Es la sentencia de control iterativa fundamental en PASCAL. Repite la ejecución de una sentencia mientras se cumple una condición específica. La sentencia que se repite puede ser compuesta.

Sintaxis: WHILE expresión\_lógica DO sentencia



La secuencia detallada de operaciones es la siguiente:

1. Evaluar la condición. Si es FALSE, seguir con el paso 4.
2. Ejecutar la sentencia componente, sabiendo que la condición es TRUE.
3. Volver al paso 1.
4. Sabiendo que la condición es FALSE, seguir con el resto del programa.

Notar que la expresión se evalúa antes de ejecutar la sentencia. Si el valor inicial es FALSE, no se ejecutará la sentencia ninguna vez.

La sentencia que se repite debe modificar el valor de la expresión, si no resulta un bloque sin salida.

Para ejecutar un grupo de sentencias, hay que construir una sentencia compuesta con los delimitadores BEGIN y END.

Ejemplo 1: Esta sentencia hace avanzar hasta el fin de fichero

```
WHILE NOT EOF (fichero) DO
    READLN (fichero);
```

Ejemplo 2: En este caso se lee caracter a caracter en la línea actual y se chequea cada uno. Si no es letra ni dígito, se incrementa un contador de errores.

```
WHILE NOT EOLN DO
    BEGIN
        READ (x);
        IF NOT (x IN ['A'..'Z', 'a'..'z', '0'..'9'])
            THEN
                error := error+1;
    END.
```

Ejemplo 3: Este fragmento de programa calcula la potencia N-ésima de x para valores no negativos de N.

```
VAR
    x, N, i, potencia : INTEGER;
...
potencia := 1; i := 0;
WHILE i < N DO
    BEGIN
        potencia := potencia * x;
        i := i+1
    END
```

Ejemplo 4: Aquí se leen valores reales por INPUT hasta alcanzar el fin de fichero, se acumulan y se escribe el valor de la media aritmética.

```

PROGRAM Valor_medio (INPUT, OUTPUT) ;
(* Ejemplo PAS006
  lee valores reales por INPUT y calcula la media aritmética *)
VAR
  num : INTEGER ;
  n, suma, media : REAL ;
BEGIN
  suma := 0 ; num := 0 ;
  WHILE NOT EOF DO
    BEGIN
      WHILE NOT EOLN DO
        BEGIN
          READ (n) ; num := num+1 ; suma := suma+n
        END ;
        READLN ;
      END ;
      media := suma / num ;
      WRITELN ('La media aritmética es : ', media)
    END.

```

Ejemplo 5: Con este caso se intenta llamar la atención sobre una confusión que suele presentarse entre los programadores principiantes.

```

PROGRAM Escribe_enteros (OUTPUT) ;
VAR
  n : INTEGER ;
BEGIN
  n := 0 ;
  WHILE n <= 10 DO
    BEGIN
      n := n+1 ;
      WRITELN (n)
    END
  END.

```

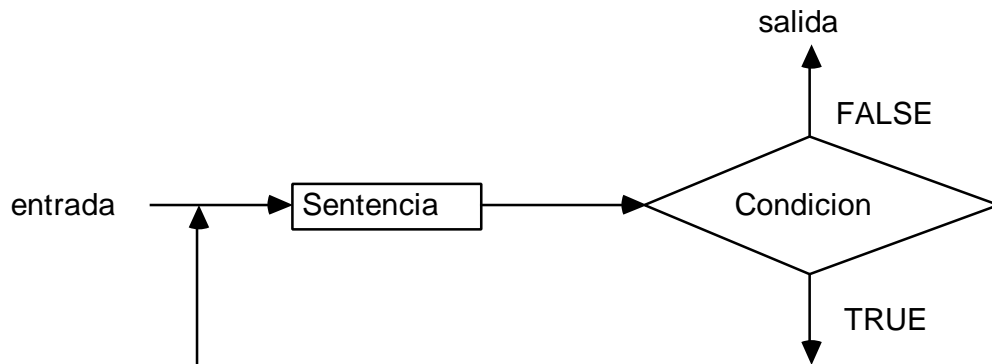
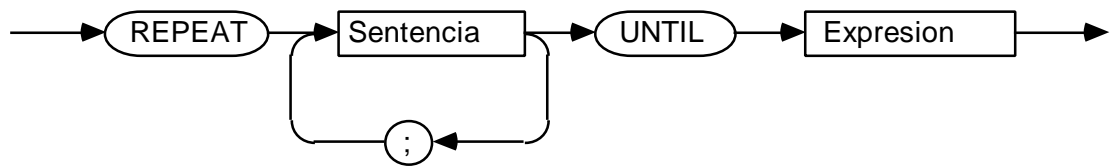
El resultado es escribir enteros desde el 1 hasta el 11. Notar que la condición de WHILE se chequea antes de ejecutar la sentencia y sólo en ese momento. Aquí, al comienzo de la última iteración, la variable n tiene el valor 10 y se ejecuta toda la sentencia compuesta.

No se debe pensar que WHILE va vigilando el valor de n y finaliza tan pronto como n=10. No es así.

### 6.5.2. La sentencia REPEAT

Ejecuta una o más sentencias hasta que se cumple una condición.

Sintaxis:        REPEAT  
                  sentencia ; ...  
                  UNTIL expresión\_lógica.



Entre las palabras reservadas REPEAT y UNTIL se pueden escribir varias sentencias sin necesidad de utilizar los delimitadores BEGIN y END. Aquí actúa UNTIL como indicador de final de rango.

En la ejecución, la expresión es evaluada después de que han sido ejecutadas las sentencias; por tanto, el rango del bucle se ejecuta una vez, por lo menos, siempre.

Ejemplo 1: Con el programa propuesto a continuación se pretende conocer el número de términos de la serie que es necesario tomar para satisfacer la desigualdad:

$$1 + 1/2 + 1/3 + \dots + 1/n > \text{límite.}$$

```

PROGRAM Serie (INPUT, OUTPUT) ;
(* Ejemplo PAS008
  Obtiene el número de términos de la serie que es necesario tomar
  para satisfacer la desigualdad:
       $1 + 1/2 + 1/3 + \dots + 1/n > \text{límite.} *$ )
VAR      contador : INTEGER ;
         suma, limite : REAL ;
BEGIN
  contador = 0 ;
  suma = 0.0 ;
  READLN (limite) ;
  REPEAT
    contador := contador+1 ;
    suma := suma+1 / contador
  UNTIL suma > limite ;
  WRITELN (contador)
END.

```

Ejemplo 2: Se propone un programa para calcular la raíz cuadrada de un número con una precisión conocida, usando el método de aproximación de Newton que establece:

- Si  $A$  es una aproximación de la raíz cuadrada de  $N$ , entonces  $(N / A + A) / 2$  es otra aproximación mejor.

El programa usa el valor 1 como primera aproximación y realiza iteraciones hasta obtener un valor de la raíz cuadrada suficientemente preciso.

Como condición para terminar las iteraciones se usa la siguiente:

$|N / A^2 - 1| < 10^{-6}$  que garantiza precisión de algunas partes por millón independientemente de la magnitud de  $N$ .

```

PROGRAM Raiz_cuadrada_aprox (INPUT, OUTPUT) ;
(* Ejemplo PAS007
  Obtiene la raíz cuadrada por el método de aproximación de
  Newton *)
CONST   epsilon = 1E-6 ;
VAR     n, raiz : REAL ;
BEGIN
  WRITELN (' Numero? '); READLN (n) ;
  IF n < 0
  THEN  WRITELN ('Error. Número negativo')
  ELSE  IF n=0.0
        THEN WRITELN (0)
        ELSE BEGIN
              raiz := 1 ;
              REPEAT
                raiz := (n / raiz+raiz) / 2
              UNTIL ABS (n / SQR (raiz) - 1) < epsilon
              WRITELN (raiz)
            END
  END.

```

### 6.5.3. La sentencia FOR

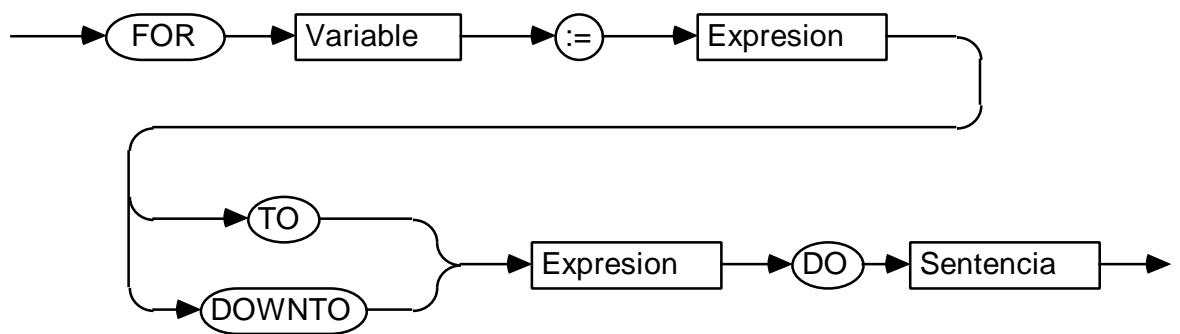
Permite construir bucles repetitivos controlados por un contador.

Con FOR se especifica la ejecución repetitiva de una sentencia según el valor de una variable de control que se incrementa o decrementa automáticamente.

Sintaxis:

TO

FOR variable\_control := valor\_inicial DOWNTO valor\_final DO sentencia



La variable de control, el valor inicial y el valor final deben ser todos del mismo tipo ordinal. Las sentencias que se repiten (rango del bucle) no deben modificar el valor de la variable de control.

El test de final de bucle se realiza antes de ejecutarse la sentencia FOR; por tanto, es posible que el bucle no se ejecute ninguna vez.

En la forma TO, si la variable de control tiene un valor menor o igual que el valor final, se ejecuta el bucle y se incrementa el valor de la variable de control. Cuando el valor de la variable de control es mayor que el valor final, se da por terminada la ejecución del bucle.

La forma DOWNTO es semejante a la forma TO, evolucionando en sentido decreciente.

La variable de control se incrementa o decrementa en unidades del tipo apropiado. Para variables de tipo INTEGER se añade o se resta una unidad en cada iteración. Para otros tipos, la variable toma un sucesor o predecesor.

Cuando el bucle termina de forma natural (se completa), el valor de la variable de control queda indefinido. Si termina por la acción de una sentencia GOTO, antes de completarse, la variable de control retiene el último valor asignado.

La sentencia FOR proporciona el modo más natural para operar con "arrays".

Ejemplo 1: Aquí se escribe una lista de los años bisiestos en el siglo 19.

```
FOR anio := 1899 DOWNTO 1801 DO
  IF (anio MOD 4) = 0
  THEN
    WRITELN (anio, ' es un año bisiesto ');
```

Ejemplo 2: Producto matricial

```
VAR
  A : ARRAY [1..20, 1..30] OF REAL ;
  B : ARRAY [1..30, 1..40] OF REAL ;
  C : ARRAY [1..20, 1..40] OF REAL ;
...
FOR i := 1 TO 20 DO FOR k := 1 TO 40 DO
  BEGIN
    C [i, k] = 0.0;
    FOR j := 1 TO 30 DO C [i, k] := C[i, k]+A [i, j] * B [j, k]
  END ;
```

Ejemplo 3: Buscar en un "array" el menor índice de un componente con valor x, mediante una inspección secuencial.

```
VAR
  a : ARRAY [1..n] OF REAL ;
  x : REAL ;
...
BEGIN
...
  i := 0 ;
  REPEAT
    i := i+1
  UNTIL ( a [i] = x ) OR ( i = n ) ;
  IF a [i] <> x THEN WRITELN ('Ese elemento no está en a')
    ELSE WRITELN ('Índice menor :', i )
END.
```

Ejemplo 4: El mismo tipo de búsqueda se puede acelerar si los elementos están ordenados en el "array". En este caso se usa frecuentemente el método de "búsqueda binaria" o "bisección" que consiste en dividir repetidamente el intervalo de búsqueda en partes iguales. Así el número de comparaciones necesario es  $\log_2(N)$  como máximo.

```
i := 1 ; j := N ;
REPEAT
  k := ( i+j ) DIV 2 ;
  IF x > a [k] THEN i := k+1 ELSE j := k-1
UNTIL ( a [k] = x ) OR ( i > j )
```

## 6.6. LA SENTENCIA WITH

Permite utilizar una notación abreviada para hacer referencias a los campos de las variables de tipo RECORD.

Sintaxis: WITH variable, ... DO sentencia

Se puede indicar una o más variables de tipo RECORD a las que se hace referencia en la sentencia.

Dentro de la sentencia WITH se pueden hacer referencias a los campos de un registro escribiendo sólo el nombre del campo.

Cuando se especifica más de una variable, el efecto es el mismo que si se anidan varias sentencias WITH. Si los registros a los que se alude están anidados, sus nombres deben aparecer en el mismo orden que en la declaración de los tipos; si no están anidados, pueden aparecer en cualquier orden.

Ejemplo:

```

TYPE  nombre = STRING [32] OF CHAR ;
      fecha = RECORD
          dia : 1..31 ;
          mes : 1..12 ;
          anio : INTEGER
      END ;
VAR   ficha : RECORD
          paciente : nombre ;
          fecha_nacimiento : fecha
      END ;
...
WITH ficha, fecha_nacimiento DO
    BEGIN
        paciente := 'PEPE PEREZ' ;
        dia := 5 ;
        mes := 9 ;
        anio := 1960
    END ;

```

que es equivalente a :

```

WITH ficha DO
    WITH fecha_nacimiento DO
        BEGIN
            paciente := 'PEPE PEREZ' ;
            dia := 5 ;
            mes := 9 ;
            anio := 1960
        END ;
    END ;

```

### 6.7. La sentencia GOTO

La sentencia GOTO produce la bifurcación incondicional del control a una sentencia identificada por una etiqueta específica.

Sintaxis: GOTO etiqueta

La sentencia GOTO debe estar dentro del alcance de declaración de la etiqueta.

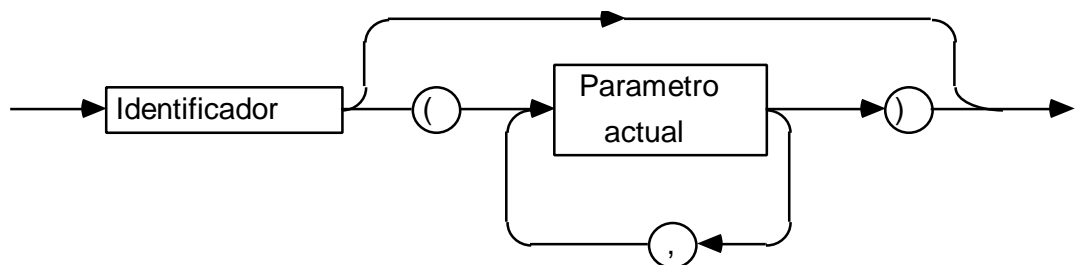
Con GOTO no se puede saltar al interior de una sentencia estructurada desde fuera.

En los tratados sobre PASCAL se recomienda vivamente NO USAR esta sentencia porque rompe la estructura de los programas. En algunas compiladores no está implementada.

### 6.8. La sentencia LLAMADA A PROCEDIMIENTO

La llamada a un procedimiento hace que se ejecute pasándole los parámetros actuales que serán asociados con los parámetros formales existentes en la declaración. Cuando termina la ejecución del procedimiento, el control vuelve a la sentencia que sigue a la llamada.

Sintaxis: Identificador\_de\_rutina (parámetro, ...)



Como parámetro se puede escribir una expresión del tipo apropiado, o el nombre de una función o el de un procedimiento.

En un capítulo posterior, dedicado a las rutinas, se tratará la problemática que presenta la comunicación con los procedimientos.

## **7. LAS RUTINAS. PROCEDIMIENTOS Y FUNCIONES.**

Cuando se presenta la necesidad de ejecutar un mismo grupo de sentencias en distintos lugares del programa, conviene disponer de algún recurso para no tener que escribirlas repetidamente, lo cual resulta tedioso y presenta otros inconvenientes.

La solución que se suele ofrecer consiste en agrupar cada uno de esos conjuntos de sentencias destacándolos y asignándoles un nombre, constituyendo así las llamadas RUTINAS o subprogramas. Luego, basta con invocar su nombre en cualquier sentencia para conseguir que se ejecute todo el conjunto.

En PASCAL, se pueden manejar dos tipos de rutinas: los PROCEDIMIENTOS y las FUNCIONES.

Pero la utilización de subprogramas se considera de forma especial en el PASCAL, lenguaje enfocado a conseguir programas bien estructurados. Con las rutinas se dispone de una estructura muy apropiada para practicar el método de desarrollo en etapas sucesivas que consiste en abordar el diseño de los programas descomponiendo el problema en partes más sencillas y repitiendo este proceso sucesivamente hasta que la solución de cada una de las partes sea trivial. Entonces resulta fácil desarrollar cada uno de los elementos del programa, depurarlos y probarlos independientemente hasta que funcionen correctamente.

Con ese enfoque, el programa queda organizado en módulos que se expresan en forma de rutinas. Una recomendación muy extendida propone que la escritura de cada subprograma no sobrepase el tamaño de un folio para facilitar su lectura y comprensión.

En definitiva, el diseño modular va dirigido a conseguir programas fiables y legibles:

- El esfuerzo de la programación se concentra finalmente en subproblemas pequeños y sencillos
- Es fácil escribir módulos pequeños libres de error y verificarlos independientemente con datos simulados
- Los listados pequeños son más legibles
- Resulta natural la adaptación al trabajo en equipo. Las tareas de desarrollar los módulos se pueden repartir entre varias personas.

El interés de utilizar subprogramas también se justifica por la construcción de bibliotecas de rutinas de utilidad general que puedan ser invocadas desde los programas que escribe el usuario para resolver casos particulares.

En PASCAL, se ofrece además una posibilidad interesante: las rutinas pueden llamarse a sí mismas (RECURSIVIDAD); lo que permite plantear algoritmos simples para resolver algunos problemas cuya solución es difícil con los medios tradicionales de la SECUENCIA, las SENTENCIAS CONDICIONALES y la ITERACION.

## 7.1 CONCEPTOS

Una rutina es un conjunto de sentencias que tienen asociado un identificador y se ejecutan como un grupo cuando se le invoca desde la sección ejecutable del programa.

En las rutinas se pueden utilizar y modificar variables del resto del programa en determinadas condiciones.

Hay algunos procedimientos y funciones que están predefinidos por el compilador (READ, WRITE, EOLN, EOF, NEW, etc...); pero el usuario puede definir otros declarándolos de forma semejante a como se hace con otras entidades (constantes, tipos, variables).

Las FUNCIONES se distinguen de los procedimientos en que devuelven un valor único asociado a su identificador, del tipo correspondiente a la declaración.

Las rutinas tienen una estructura análoga a la del programa. Se componen de:

- . el ENCABEZAMIENTO
- . el CUERPO, con:
  - . Sección de declaraciones
  - . Sección ejecutable

En realidad, las rutinas son una especie de programas. Son SUBPROGRAMAS que admiten todo lo expuesto en los capítulos anteriores referente a estructuras de datos y de control; y que pueden incluir a otros subprogramas a su vez, configurando una estructura donde el PROGRAMA completo se contempla como un conjunto de MODULOS o BLOQUES anidados que pueden compartir e intercambiarse valores.

En este esquema, el programa principal y cada uno de los subprogramas constituyen ámbitos de acción diferentes donde se manejan entidades como constantes, etiquetas, tipos, variables, que deben haber sido declaradas previamente.

La parte del programa donde se tiene acceso a un identificador se llama ámbito o campo de existencia. Fuera de ese dominio, un identificador no tiene significado o tiene significado diferente. La regla general establece que **un identificador es conocido en el bloque donde ha sido declarado y en todos los que tiene anidados.**

### **Entidades globales, locales y estandar**

A las entidades que se declaran en el programa principal se les llama GLOBALES (variables globales, por ejemplo). Son conocidas a lo largo de todo el programa y existen durante toda la ejecución, si corresponden a estructuras estáticas.

A las entidades que sólo tienen significado en una parte del programa se les llama LOCALES. Son las que se declaran en un procedimiento o función y son conocidas sólo en el mismo y en cualquier otro que esté anidado dentro. Estos objetos sólo tienen la existencia asegurada mientras dura la ejecución del procedimiento donde han sido declaradas, después se libera el espacio que ocupaban en memoria quedando disponible para ser utilizado por otras.

En tercer lugar, se habla de objetos o entidades STANDARD, refiriéndose a los que están definidos en el lenguaje (compilador). Obviamente, éstos son conocidos en cualquier parte del programa.

En estas condiciones, queda claro que en un procedimiento se pueden manejar cualquier objeto GLOBAL y todos los que hayan sido declarados localmente; y que es necesario disponer de algún mecanismo para hacer posible la comunicación entre procedimientos diferentes. Esto se consigue mediante los PARAMETROS.

En lo que sigue, se van a tratar todos estos aspectos referentes al uso de las rutinas, considerando:

- . Cómo se declaran
- . Cómo funcionan los mecanismos para la comunicación entre módulos mediante parámetros
- . Cómo se invocan

```
PROGRAM Programa (INPUT, OUTPUT);  
  CONST ...  
  TYPE ...  
  VAR ...  
  
  PROCEDURE Rutina_1 .....  
    CONST ...  
    TYPE ...  
    VAR ...  
  
    PROCEDURE Rutina_2 ...  
      CONST ...  
      TYPE ...  
      VAR ...  
      BEGIN  
        Sentencias ejecutables de Rutina_2  
      END;  
  
    BEGIN  
      Sentencias ejecutables de Rutina_1  
    END;  
  
  PROCEDURE Rutina_3 ...  
    CONST ...  
    TYPE ...  
    VAR ...  
    BEGIN  
      Sentencias ejecutables de Rutina_3  
    END;  
  
  BEGIN  
    Sentencias ejecutables de Programa  
  END;
```

## 7.2 LA DECLARACION

Como cualquier otra entidad definida por el usuario, los procedimientos (PROCEDURE) y las funciones (FUNCTION) deben declararse antes de ser utilizados. En la declaración se especifican todas las partes que los forman:

- . encabezamiento
- . declaraciones
- . sentencias ejecutables

En la sección de las declaraciones de un procedimiento o función, se pueden declarar otros subprogramas.

Sintaxis:           PROCEDURE identificador (lista de parámetros formales);  
                          Sección de las declaraciones;  
                  BEGIN  
                          Sección ejecutable  
                  END;

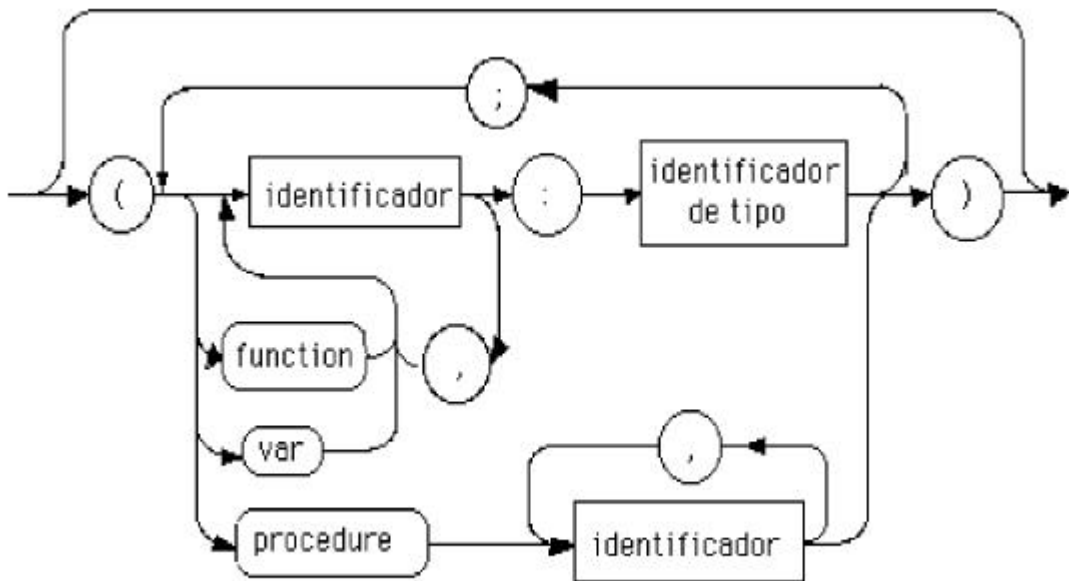
                  FUNCTION identificador (lista de parámetros formales) : Tipo ;  
                          Sección de las declaraciones;  
                  BEGIN  
                          Sección ejecutable  
                  END;

Identificador: da nombre a la rutina y se utiliza para invocarla al solicitar su ejecución. Se construye siguiendo las mismas reglas que rigen para las constantes y las variables.

Lista de parámetros formales : especificación de los identificadores que se utilizan para comunicar valores con otros módulos del programa. Se indican sus tipos y los mecanismos para la transferencia.

Sección de las declaraciones : contiene las declaraciones de todas las entidades definidas por el usuario que se utilizan dentro de la rutina: etiquetas, constantes, tipos, variables, procedimientos, funciones. Son identificadores locales.  
No se admite la inicialización de variables.  
No se puede volver a declarar los nombres de los parámetros formales.

Sección ejecutable : contiene las sentencias con las que se realizan las acciones que constituyen el subprograma. Termina con el delimitador END seguido de punto y coma.



### Lista de parámetros formales

En la declaración de una FUNCION es necesario especificar el tipo al que pertenece, lo que equivale a determinar los valores permitidos que puede llevar asociados el identificador de función. Sólo se admiten los tipos escalares y los punteros.

A continuación se muestran algunos ejemplos:

Ejemplo 1. Se leen caracteres por INPUT y se almacenan en una tabla. Incrementando un contador se conoce el número de caracteres leídos. En este caso no se han declarado parámetros. Se supone que "lon", "lonmax" y "cadena" son variables globales. No se han declarado objetos locales.

```
PROCEDURE Leer_cadena;
BEGIN
    lon:=0;
    WHILE (NOT EOLN) AND (lon<lonmax) DO
        BEGIN
            lon := lon+1;
            READ (cadena [lon] )
        END;
    READLN
END;
```

Ejemplo 2. La función "Circulo" es de tipo REAL. Cuando se ejecute, se obtendrá un valor REAL asignado al identificador "Circulo". El parámetro formal es "radio"

```
FUNCTION Circulo ( radio: REAL ) : REAL;
  CONST pi = 3.1416;
  BEGIN
    Circulo := pi * radio * radio
  END;
```

Ejemplo 3. Aquí se leen valores por INPUT y se construye un conjunto con ellos. El tipo "dia" tiene que estar definido antes. La variable "dia\_libre" es local.

```
PROCEDURE Leer_conjunto ( VAR dias : SET OF dia);
  VAR dia_libre : dia;
  BEGIN
    dias := [];
    WHILE NOT EOLN DO
      BEGIN
        READ (dia_libre);
        dias := dias + [dia_libre]
      END;
    READLN
  END;
```

Ejemplo 4. Esta función busca al menor de los elementos de una tabla recorriendola entera. Los tipos "tablas" y "elemento" deben haber sido declarados antes. El tipo "tablas" tiene que ser ARRAY ...OF elemento. La variable "i" es una variable local.

```
FUNCTION Menor ( tabla : tablas , num : INTEGER ) : elemento;
  VAR i : INTEGER;
  BEGIN
    Menor := tabla [1];
    FOR i:=2 TO num DO IF tabla[i]<Menor THEN Menor:=tabla[i]
  END;
```

### 7.3 LOS PARAMETROS. MECANISMOS DE SUSTITUCION

En una rutina se puede manejar cualquier entidad GLOBAL.

Cuando el único objetivo que se persigue con la utilización de rutinas es segmentar el programa en módulos más pequeños para facilitar su legibilidad, no hay demasiados inconvenientes en manejar objetos de carácter global desde el interior de los subprogramas.

Pero el interés de utilizar rutinas se extiende y crece en importancia en los casos frecuentes donde hay que repetir la ejecución de grupos de sentencias a lo largo del programa, en contextos distintos y con valores iniciales diferentes. Entonces no conviene actuar sobre entidades globales desde el interior de los subprogramas para no correr el riesgo de modificar inadvertidamente valores externos al procedimiento o función. Es preferible actuar con entidades de ámbito local y disponer de alguna vía para relacionarlas con las definidas a nivel global y con las que son locales de otros subprogramas.

Hay ocasiones en las que la relación entre una rutina y el programa principal trasciende al caso de un programa determinado. Es cuando se escriben rutinas de utilidad general que se redactan de forma que cualquier usuario las pueda incluir en su programa.

En todos estos casos se necesita disponer de algún mecanismo para intercambiar valores entre las rutinas y el resto del programa. Ello se consigue mediante las listas de PARAMETROS, llamados también ARGUMENTOS.

Los parámetros que se indican en la declaración de las rutinas se llaman PARAMETROS FORMALES. Son identificadores de entidades locales con las que se pueden introducir y sacar valores en las rutinas. En las funciones, los parámetros formales sólo permiten la entrada, puesto que la salida produce un valor único que va asociado al identificador de la propia función.

Los parámetros que se indican en la llamada a las rutinas son los PARAMETROS ACTUALES.

```
Ejemplo:      ...
                PROCEDURE Altas ( codigo:articulos, cantidad:INTEGER,
                                fecha:fechas, seccion:CHAR);
                ...
                Leer_datos;
                Altas (datos.codigo, datos.cantidad, (12,3,1987), 'C');
                ...
```

En el ejemplo, "codigo", "cantidad", "fecha" y "seccion" son parámetros formales del procedimiento "Altas". Los parámetros actuales que se pasan en la llamada al procedimiento son "datos.codigo", "datos.cantidad", (12,3,1987), 'C'.

Cuando se ejecuta una llamada a rutina, se establece una asociación entre los parámetros actuales y los formales que permanece activa durante toda la ejecución del subprograma, haciendo posible el intercambio de valores con entidades locales de la rutina.

```
Ejemplo:      ...
                VAR          r, radio, circulo, esfera : REAL;
                FUNCTION Potencia ( base:REAL, exponente:INTEGER):REAL;
                VAR          i : INTEGER;
                BEGIN
                    Potencia := 1;
                    FOR i:=1 TO exponente DO Potencia:=Potencia*base
                END;
                ...
                circulo := 3.1416 * Potencia (radio, 2);
                esfera := 4/3 * 3.1416 * Potencia (r, 3);
```

En el ejemplo, "base" y "exponente" son los argumentos de la función "Potencia". Son sus parámetros formales. La variable "i" es una variable local de "Potencia". Las variables "radio", "r" y las constantes "2" y "3" son los parámetros actuales que se pasan en las dos sentencias de llamada. Al ejecutarse la primera llamada para calcular la superficie de un círculo, se pasan los valores de "radio" y "2". Esos valores se asignan a los parámetros formales correspondientes, "base" y "exponente", y se realiza el cálculo devolviendo un valor de tipo REAL asignado al identificador "Potencia". Más adelante, se invoca a la misma función pasándole unos valores diferentes.

La correspondencia entre los parámetros formales y los actuales se establece por la posición que ocupan en las listas, y la sustitución se realiza aplicando algunos de los mecanismos siguientes: por VALOR, por REFERENCIA, por NOMBRE. Según el mecanismo que actúa, se habla de varias clases de parámetros:

### **Parámetros-valor**

En la sustitución por valor, se evalúa el parámetro actual y el valor resultante se asigna al parámetro formal correspondiente. El parámetro actual se puede expresar como una constante, una variable o una expresión.

El uso de un parámetro-valor consiste en la mera transferencia de un valor a la rutina. Estos son parámetros de entrada. Se pueden modificar dentro de la rutina, pero ello no afecta al parámetro actual correspondiente. No permiten sacar información de la rutina.

Los parámetros-valor se declaran indicando simplemente su nombre y su tipo, sin indicar prefijos. La ausencia de prefijos es precisamente lo que identifica a esta clase de parámetros.

Ejemplos:           FUNCTION Circulo (radio : REAL ) : REAL;  
                          ...  
                          FUNCTION Menor ( tabla: tablas, num : INTEGER):elemento;

### **Parámetros-variable**

Permiten aplicar el mecanismo de sustitución POR REFERENCIA, por el que se produce una asociación entre parámetro actual y parámetro formal que consiste en una especie de sustitución. En realidad se asignan al identificador de parámetro formal las mismas direcciones de memoria que corresponden al parámetro actual, de manera que cualquier modificación que se produzca dentro del procedimiento afecta a los parámetros actuales que son entidades externas al mismo.

Para describir gráficamente la situación, se puede imaginar una flecha para cada parámetro formal, que una su nombre con la dirección de memoria donde está almacenado su parámetro actual correspondiente. Toda operación que afecte al parámetro formal, en realidad se realiza sobre el parámetro actual.

Con este mecanismo de sustitución por referencia, se puede conseguir la transferencia de información con las rutinas en los dos sentidos: entrada y salida.

Los parámetros-variable se declaran precedidos de la palabra reservada VAR. Los parámetros actuales correspondientes tienen que ser variable del mismo tipo.

Con los ejemplos que se muestran a continuación, se pueden notar las diferencias entre los mecanismos de paso POR VALOR y POR REFERENCIA, por los efectos que producen.

```

PROGRAM P;
  VAR i : INTEGER;
      a : ARRAY [1..2] OF INTEGER;
  PROCEDURE Paso_por_valor ( x : INTEGER);
  BEGIN
      i := i+1; x := x+2
  END;
BEGIN
  a[1] := 10; a[2] := 20;
  i := 1;
  Paso_por_valor ( a[i] )
END.

```

Al ejecutarse el procedimiento, "x" es una variable local, de tipo INTEGER, que toma inicialmente el valor "10". Después se incrementa su valor en dos unidades, pero ello no afecta a la variable global "a". Después de que termine la ejecución del procedimiento, la variable "a" seguirá teniendo el mismo valor que antes ( a=(10,20)) y la variable "x" habrá dejado de existir.

```

PROGRAM P;
  VAR i : INTEGER;
      a : ARRAY [1..2] OF INTEGER;

  PROCEDURE Paso_por_referencia ( VAR x : INTEGER );
  BEGIN
      i := i+1; x := x+2
  END;
BEGIN
  a [1] := 10; a [2] := 20;
  i :=1 ;
  Paso_por_referencia ( a[i] )
END.

```

Aquí la sustitución se realiza sobre el primer elemento a[1]. La sentencia x:=x+2 significa realmente : a[1] := a[1]+2; de manera que después de ejecutarse el procedimiento, se habrá modificado el valor de la variable "a" ( a=(12,20)).

### **Parámetros-procedimiento y parámetros-función**

En la lista de parámetros formales, también se pueden incluir funciones y procedimientos. En tal caso hay que escribir su encabezamiento completo y tales rutinas sólo admiten parámetros pasados por valor.

Así se puede conseguir que desde el interior de una rutina se pueda invocar a otras más externas, cuyo efecto depende del estado de la ejecución.

Ejemplos:

```
PROCEDURE Reducir ( FUNCTION f(x:REAL) : REAL;
VAR a, b, x1, x2, y1, y2 : REAL);
```

```
FUNCTION demo2 (PROCEDURE muestra(VAR x,y,z:REAL);
FUNCTION demo1(a,b,c : REAL) : REAL ):REAL;
```

## 7.4 LA RECURSIVIDAD

Es la capacidad de un rutina de llamarse a sí misma.

Esta técnica permite resolver de forma natural los problemas de tipo recursivo, que son difíciles de abordar con otros medios.

Al escribir una rutina recursiva es necesario incluir una condición de terminación para evitar que la recursión continúe indefinidamente.

Un ejemplo sencillo se puede plantear con una función para obtener el factorial de un número de forma recursiva:

```
FUNCTION Factorial ( n : INTEGER ) : INTEGER;
BEGIN
  IF n<= 1 THEN Factorial := 1
  ELSE Factorial := n * Factorial (n-1)
END;
```

### El problema de las torres de Hanoi

Este es un ejemplo muy típico donde la recursividad permite una formulación cómoda del problema, cuya solución resulta difícil por otras vías.

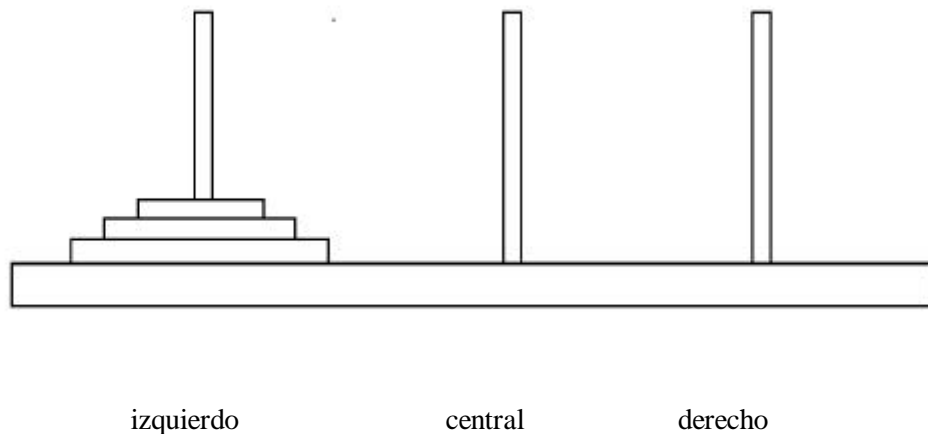
Se trata de tres postes y un conjunto de discos de varios tamaños con un orificio en el centro que permite apilarlos alrededor de los postes. Inicialmente todos los discos están apilados en el poste de la izquierda y ordenados por tamaño de forma que el mayor está situado en la base y el más pequeño está en la cima.

El objetivo es trasladar todos los discos desde el poste izquierdo al derecho sin colocar nunca un disco mayor sobre otro de menor tamaño. Sólo se puede mover un disco cada vez y cada disco debe estar colocado siempre en algún poste.

La estrategia consiste en considerar a uno de los postes como origen y a otro como destino. El tercero se utilizará como almacenamiento intermedio para permitir el traslado de los discos.

Así, si los discos se encuentran inicialmente en el poste izquierdo, el problema de trasladar "n" discos al poste derecho se puede representar:

1. Mover los n-1 discos superiores al poste central, usando el poste derecho como almacen intermedio
2. Trasladar el disco restante al poste derecho
3. Llevar los n-1 discos del poste central al poste derecho, usando el poste izquierdo como almacen intermedio



De esta manera el problema queda expresado en términos de recursividad que se puede aplicar a cualquier valor de "n" mayor que 1. Cuando  $n=1$ , la solución es trivial pues consiste simplemente en llevar el disco del poste izquierdo al derecho.

Con esta formulación, el procedimiento tendrá la estructura siguiente:

```

PROCEDURE Trasladar (n, origen, destino, otro : INTEGER);
IF n>0 THEN
    trasladar n-1 discos desde el origen al poste intermedio
    trasladar el disco restante desde el origen al destino
    trasladar n-1 discos desde el poste intermedio al destino
END;
```

La transferencia de n-1 discos puede realizarse mediante una llamada recursiva a Trasladar. De forma que se puede escribir:

Trasladar (n-1, origen, otro, destino)

para realizar la primera transferencia, y

Trasladar (n-1, otro, destino, origen)

para realizar la segunda.

El traslado de un disco desde origen al destino se puede representar mediante la escritura de los valores de origen y destino, que puede llevarse a cabo desde otro procedimiento "Moverdisco", definido dentro de "Trasladar".

En estas condiciones, el procedimiento completo queda:

```

PROCEDURE Trasladar ( n, origen, destino, otro : INTEGER);
  PROCEDURE Moverdisco (origen, destino : INTEGER);
  BEGIN
    WRITELN ('Trasladar ', origen:1, ' a ', destino:1)
  END;
BEGIN
  IF n>0 THEN BEGIN
    Trasladar (n-1, origen, otro, destino);
    Moverdisco (origen, destino);
    Trasladar (n-1, otro, destino, origen)
  END
END;

```

Ahora ya sólo queda escribir el programa principal que se encarga simplemente de leer el valor de "n" y de iniciar la operación llamando al procedimiento "Trasladar". En esta primera llamada se especificarán los parámetros actuales como números enteros que identifican a cada uno de los tres postes ("1" para el poste de la izquierda, "2" para el del centro y "3" para el de la derecha: Trasladar (n, 1, 3, 2)

```

PROGRAM Torres_de_Hanoi (INPUT, OUTPUT);
VAR n : INTEGER;
(**)
PROCEDURE Trasladar ( n, origen, destino, otro : INTEGER);
  PROCEDURE Moverdisco (origen, destino : INTEGER);
  BEGIN
    WRITELN ('Trasladar ', origen:1, ' a ', destino:1)
  END;
BEGIN
  IF n>0 THEN BEGIN
    Trasladar (n-1, origen, otro, destino);
    Moverdisco (origen, destino);
    Trasladar (n-1, otro, destino, origen)
  END
END;
(**)
BEGIN
  WRITELN ('Introduzca el número de discos: ');
  READLN (n);
  Trasladar (n, 1, 3, 2)
END.

```

Cuando se ejecute este programa para el caso de tres discos, escribirá una salida como esta

```
Trasladar 1 a 3
Trasladar 1 a 2
Trasladar 3 a 2
Trasladar 1 a 3
Trasladar 2 a 1
Trasladar 2 a 3
Trasladar 1 a 3
```

## 7.5 LAS RUTINAS PREDECLARADAS ESTANDAR

Son rutinas de utilidad general que están definidas en el lenguaje y se encuentran disponibles en cualquier lugar del programa.

Los fabricantes suelen ofrecer colecciones abundantes de rutinas predeclaradas en sus implementaciones del lenguaje sobre equipos concretos. Las que aquí se presentan son las que corresponden a la definición del lenguaje según Wirth, su autor.

Se van a presentar clasificadas en varios grupos:

- Para manejo de ficheros:  
PUT, GET, RESET, REWRITE, READ, WRITE, READLN,  
WRITELN, PAGE
- De asignación dinámica de memoria:  
NEW, DISPOSE
- De movimiento de datos:  
PACK, UNPACK
- Funciones aritméticas:  
ABS, SQR, SQRT, SIN, COS, ARCTAN, EXP, LN
- Predicados o funciones booleanas:  
ODD, EOLN, EOF
- Funciones de transferencia entre tipos:  
TRUNC, ROUND, ORD, CHR
- Otras funciones:  
SUCC, PRED

### Procedimientos para el manejo de ficheros

PUT(f) agrega el valor del buffer  $f^{\wedge}$  a la secuencia f. El efecto está definido si se cumple la condición previa de que EOF(f) sea verdadero. El valor EOF(f) permanece verdadero y el valor de  $f^{\wedge}$  queda indefinido.

- GET(f) avanza el mecanismo de posicionamiento hasta el siguiente componente de la secuencia, y asigna su valor al buffer f<sup>^</sup>. Si no existe un componente siguiente, entonces EOF(f) toma el valor verdadero y el valor de f<sup>^</sup> queda indefinido. El efecto de GET(f) sólo está definido si se cumple la condición previa de que EOF(f) sea falso.
- RESET(f) reinicializa el posicionamiento al comienzo del fichero y asigna el valor del primer elemento de f al buffer f<sup>^</sup>. El valor de EOF(f) se hace falso si f no está vacío; en caso contrario, f<sup>^</sup> quedará indefinido y EOF(f) quedará verdadero.
- REWRITE(f) descarta el valor actual de f, de forma que puede ser generada una secuencia nueva. La función EOF(f) toma el valor verdadero.
- PAGE(f) produce un salto al principio de la siguiente página de impresora, cuando el fichero f ese está imprimiendo.
- READ(f,v) asigna el valor del buffer a la variable "v", avanza el mecanismo de posicionamiento al elemento siguiente y asigna su valor al buffer.
- WRITE(f,e) asigna el valor de "e" al buffer y añade el valor del buffer al final de la secuencia.
- READLN(f) salta posiciones (caracteres) en el fichero "f", de tipo TEXT, hasta el siguiente carácter inmediato a la próxima marca de fin de línea.
- WRITELN(f) añade una marca de fin de línea al final del fichero "f" de tipo TEXT.

### Procedimientos de asignación dinámica de memoria

- NEW(p) crea una nueva variable "v" y asigna a la variable puntero "p" un puntero a la variable "v".
- DISPOSE(p) indica que el espacio de almacenamiento ocupado por la variable "p" ya no es necesario.

### Procedimientos de movimiento de datos

Teniendo en cuenta que: a : ARRAY [m..n] OF T;  
z : PACKED ARRAY [u..v] OF T

PACK(a,i,z) significa: FOR j:=u TO v DO z[j] := a[j-u+i]

UNPACK(z,a,i) significa: FOR j:=u TO v DO a[j-u+i] := z[j]

**Funciones aritméticas**

ABS(x)	calcula el valor absoluto de "x". El tipo de "x" debe ser INTEGER o REAL. El resultado es del mismo tipo que "x".
SQR(x)	calcula el cuadrado de "x". El tipo de "x" debe ser INTEGER o REAL. El resultado es del mismo tipo que "x".
SQRT(x)	calcula la raíz cuadrada de "x", siendo $x \geq 0$ . El tipo de "x" debe ser INTEGER o REAL. El resultado es de tipo REAL.
SIN(x)	calcula el seno de un ángulo "x" expresado en radianes. El tipo de "x" debe ser INTEGER o REAL. El resultado es de tipo REAL.
COS(x)	calcula el coseno de un ángulo "x" expresado en radianes. El tipo de "x" debe ser INTEGER o REAL. El resultado es de tipo REAL.
ARCTAN(x)	calcula el arco cuya tangente es "x". El tipo de "x" debe ser INTEGER o REAL. El resultado es de tipo REAL.
EXP(x)	calcula $e^x$ , siendo $e=2.7182818\dots$ la base del sistema de logaritmos neperianos. El tipo de "x" debe ser INTEGER o REAL. El resultado es de tipo REAL.
LN(x)	calcula el logaritmo neperiano de "x", siendo $x > 0$ . El tipo de "x" debe ser INTEGER o REAL. El resultado es de tipo REAL.

**Predicados o funciones booleanas**

ODD(x)	determina si "x" es par o impar. Devuelve el valor TRUE si "x" es impar, y FALSE en caso contrario. El tipo de "x" debe ser INTEGER. El resultado es de tipo BOOLEAN.
EOLN(x)	determina si se ha detectado un fin de línea. El tipo de "x" debe ser TEXT. El resultado es de tipo BOOLEAN.
EOF(x)	determina si se ha detectado un fin de fichero. El tipo de "x" debe ser FILE. El resultado es de tipo BOOLEAN.

**Funciones de transferencia entre tipos**

TRUNC(x)	suprime la parte decimal de "x". El tipo de "x" debe ser REAL. El resultado es de tipo INTEGER.
ROUND(x)	redondea el valor de "x" al entero más próximo. El tipo de "x" debe ser REAL. El resultado es de tipo INTEGER.

ORD(x) obtiene el número de orden de "x" dentro del conjunto de valores definidos por su tipo. El tipo de "x" debe ser ordinal. El resultado es de tipo INTEGER.

CHR(x) devuelve el carácter cuyo número ordinal es "x". El tipo de "x" debe ser INTEGER. El resultado es de tipo CHAR.

### **Otras funciones**

SUCC(x) devuelve el valor sucesor de "x" (el que corresponde al siguiente ordinal). El tipo de "x" debe ser ordinal. El resultado es del mismo tipo que "x".

PRED(x) devuelve el valor predecesor de "x" (el que corresponde al ordinal anterior). El tipo de "x" debe ser ordinal. El resultado es del mismo tipo que "x".

## A. 1 BIBLIOGRAFIA

1. Findlay, W.; Wat, J.A., "PASCAL. Programación metódica", Ed. Rueda, 1984.
2. Gottfried, B.S., "Programación en PASCAL", McGraww Hill, 1986.
3. Grogono, P., "Programming in PASCAL", Addison-Wesley, 1978.
4. Jensen, K.; Wirth, N., "PASCAL. Manual del usuario e informe", 2a. ed. , Ed. El Ateneo, 1985.
5. Joyanes, L., "Metodología de la programación.Diagramas de flujo, Algoritmos y Programación Estructurada" , MacGraw Hill, 1987.
6. Keller, A.M., "Programación en PASCAL", McGraw Hill, 1983.
7. Levine, G.,"Introducción a la Computación y a la Programación Estructurada", McGraw Hill, 1984.]
8. Schneider, G.M.; Weingart, S.W.; Perlman, D.M., "An introduction to programming and problem solving with PASCAL", 2a. ed., John Wiley & Sons, 1982.
9. Wirth, N., "Algoritmos + Estructuras de Datos = Programas", Ed. del Castillo, 1980.
10. Wirth, N., "Introducción a la programación metódica", 2a. ed. , Ed. El Ateneo, 1986.